Easy Java/Javascript Simulations Manual

Wolfgang Christian
and
Francisco Esquembre

Chapter One

Installing and running Easy Java/Javascript Simulations

Machines should work. People should think. *Richard Hamming*

This introductory chapter provides an overview of *Easy Java/Javascript Simulations* (*EjsS* for short), the high-level modeling and authoring tool that we use to teach modeling and simulation. We first describe the installation process and the file structure of our program, and then run the *EjsS* console to get our first *EjsS* program on the screen. We describe how *Easy Java/Javascript Simulations* supports different programming languages for the modelling process. Subsequent chapters in this document provide a step-by-step introduction to the different parts of the *EjsS* graphical user interface, as well as to the modelling process. We finally instruct you how to download existing models created with *EjsS* from our on-line digital libraries.

1.1 ABOUT Easy Java/Javascript Simulations

Computer modeling is intimately tied to computer simulation. A model is a conceptual representation of a physical system and its properties and modeling is the process whereby we construct this representation. Computer modeling requires (1) a description and an analysis of the problem, (2) the identification of the variables and the algorithms, (3) the implementation on a specific hardware-software platform, (4) the execution of the implementation and analysis of the results, (5) refinement and generalization, and (6) the presentation of results. A computer simulation is an implementation of a model that allows us to test the model under different conditions with the objective of learning about the model's behavior. The applicability of the results of the simulation to those of the real (physical) system depends on how well the model describes reality. The process of devising more general and more accurate models is what science is about.

The implementation of a model and the visualization of its output requires that we program a computer. Programming can be fun, because it gives us complete control of every visual and numerical detail of the simulated world. But programming is also a technical task that can intimidate. This technical barrier can, however, be lowered if we use an appropriate tool. *Easy Java/Javascript Simulations* is a modeling tool that has been designed to allow scientists, not only computer scientists, to create simulations in different programming languages. [1] *EjsS* simplifies this task, both from the technical and from the conceptual point of view.

EjsS provides a simple yet powerful conceptual structure for building simulations. The tool offers a sequence of workpanels which we use to implement the model and its graphical user interface. *EjsS* automates tasks such as numerically solving ordinary differential equations, and animation (using separate threads, if required). The low-level communication between the program and the end-user that takes place at run-time, including handling of mouse actions within the simulation's graphical interface, is accomplished without low-level programming.

Obviously, part of the task still depends on us. You are responsible for providing a model for the phenomenon and for designing and selecting an output view that shows the model's main features. These high-level tasks are more related to science than to programming. You are encouraged to devote your time and energy studying the science, something that the computer cannot do. The purpose of this document is to demonstrate that this computer modeling is not only possible for non-programmers, but can be relatively easy, with the help of *Easy Java/Javascript Simulations*.

1.2 INSTALLING AND RUNNING THE SOFTWARE

Let us begin by installing *Easy Java/Javascript Simulations* and running it. *EjsS* is a Java program that can be run under any operating system that supports a Java Virtual Machine (VM). Because Java is designed to be platform independent, the *EjsS* user interface on Mac OS X, Unix, and Linux look very much the same, though they may present small differences among them. (We display the Mac OS X interface in the figures of this document.)

If *Easy Java/Javascript Simulations* is not installed in your computer, you need to do so now. To **install** *EjsS*, do the following:

1. **Install the Java Runtime Environment.** *EjsS* requires the Java

[1] Currently, *EjsS* supports the Java and Javascript programming languages.

Runtime Environment (JRE), **version 1.7 or later**. The JRE may already be installed in your computer, but, if not, visit the Java site at <*http://java.com*> and follow the instructions there to download and install the latest version.

2. **Copy the *EjsS* distribution file to your computer.** *EjsS* is distributed in a compressed ZIP file that can be downloaded from *EjsS* web site <*http://www.um.es/fem/EjsWiki*>. The distribution file will be called something like **EJS_X.x_yymmdd.zip**. Here, the `X.x` characters stand for the actual version of the software, and `yymmdd` stands for the date this version was created. (For instance, you can get something like **EJS_5.1_141122.zip**.)
3. **Uncompress *EjsS*.** Uncompress the *EjsS* distribution file on your computer's hard disk to create a directory called **EJS_X.x** (**EJS_5.1** in the example). The new directory contains the whole *EjsS* program.

   In Unix-like systems, the **EJS_X.x** directory may be uncompressed as read-only. Enable write permissions for the **EJS_X.x** directory and all its subdirectories.

And that's it! This is all that is needed to install *EjsS*. Once *Easy Java/Javascript Simulations* is installed in your computer, do the following to **run** *EjsS*:

Run the *EjsS* console. Inside the newly-created **EJS_X.x** directory, you will find a file called **EjsConsole.jar**. Double-click it to run the *EjsS* console shown in Figure 1.1.

If double-clicking doesn't run the console, open a system terminal window, change to the **EJS_X.x** directory, and type the command: `java -jar EjsConsole.jar`. You'll need to fully qualify the `java` command if it is not in your system's PATH.

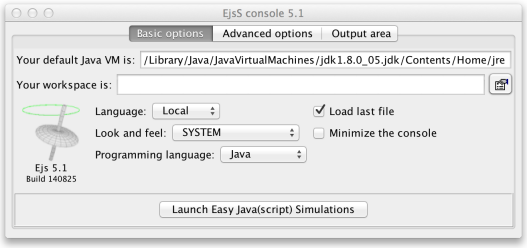

Figure 1.1: The *EjsS* console.

You should see the console (Figure 1.1) and the file chooser dialog of Figure 1.2, that we will describe below, on your computer display.

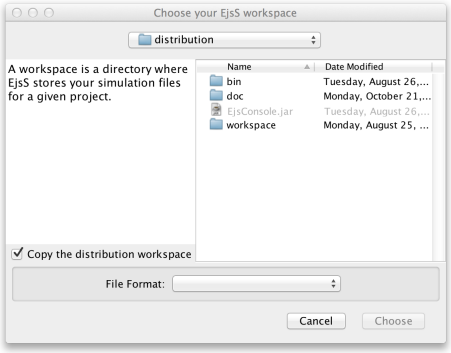

Figure 1.2: File chooser to select your workspace directory.

The *EjsS* console is not part of *EjsS*, but a utility used to launch one or several instances (copies) of *EjsS* and to perform other *EjsS*-related tasks. You can use the console to customize some aspects of how *EjsS* looks and behaves at start up. [2] The console also displays *EjsS* program information and error messages on its *Output area* tab, and we will refer to it from time to time in this document. The console creates an instance of *EjsS* at start-up and exits automatically when you close the last running instance of *EjsS*.

However, before the console can run *EjsS* right after installation, the file chooser displayed in Figure 1.2 will appear, letting you select the directory in the computer hard disk that you will use as your *workspace*. *EjsS* uses the concept of a workspace to organize your work. A workspace is a directory in your hard disk where *EjsS* stores your simulation files for a given project. A workspace can contain an unlimited number of simulations. Inside a workspace directory, *EjsS* creates four subdirectories:

- **config** is the directory for user-defined configuration and options files.
- **export** is the proposed target directory when *EjsS* generates files for distribution.
- **output** is the directory used by *EjsS* to place temporary files generated when compiling a simulation.
- **source** is the directory under which all your simulation (source and auxiliary) files must be located.

When you first run *EjsS*, the console asks you to choose a workspace directory. This must be a writable directory anywhere in your hard disk.

[2] For instance, set the *Look and feel* field to *CROSS_PLATFORM*, the specific look and feel identical across all platforms, and launch a new instance of *EjsS* to apply the change. We set the default look and feel to *SYSTEM*, the default for your platform. The one shown in this document corresponds to the characteristic *Aqua* look an feel of Mac OS X.

You can choose to use the workspace included in the distribution, i.e. the **workspace** directory in the **EJS_X.x** directory created when you unzipped the *EjsS* bundle. But it is **highly recommended** to create a new directory in your usual personal directory (or, better yet, in a cloud-accesible directory — like *Dropbox <http://dropbox.com>*, or similar). The file dialog that allows you to choose the workspace has a check box that, when checked, will copy all the examples files of the distribution to the new workspace. Leave this check box checked and you will find some subdirectories in the **source** directory of your workspace which contain sample simulations. In particular, you will find the **JavaExamples** and **JavascriptExamples** directories described in later chapters of this document.

> Although generally not needed, you can create and use more than one workspace for different projects or tasks. The console provides a selector to let you change the workspace in use and *EjsS* will remember the current workspace between sessions or even if you reinstall *EjsS*.

Finally, the first time you run *EjsS*, the program will also ask you to introduce your name and affiliation (Figure 1.3). This step is optional but recommended, since it will help you document your future simulations. You can choose to input or modify this information later using the options icon of *EjsS*' task bar.

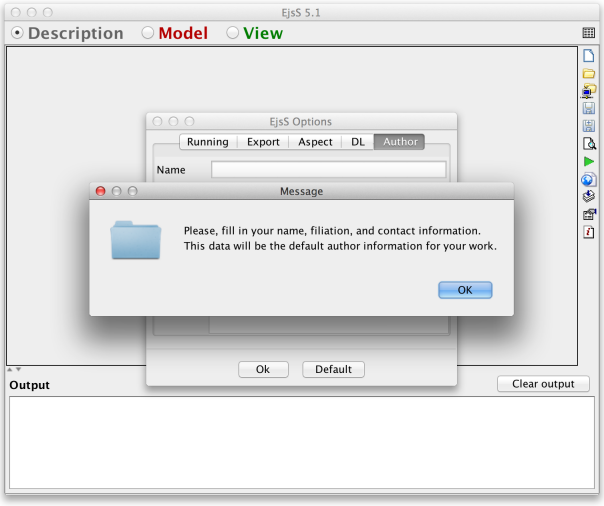

Figure 1.3: Optionally input your name and affiliation.

1.3 THE GRAPHICAL USER INTERFACE

We are now ready to turn our attention to the *EjsS* modeling tool, displayed with annotations in Figure 1.4. Despite its simple interface, *EjsS* has all the

tools needed for a complete modeling cycle.

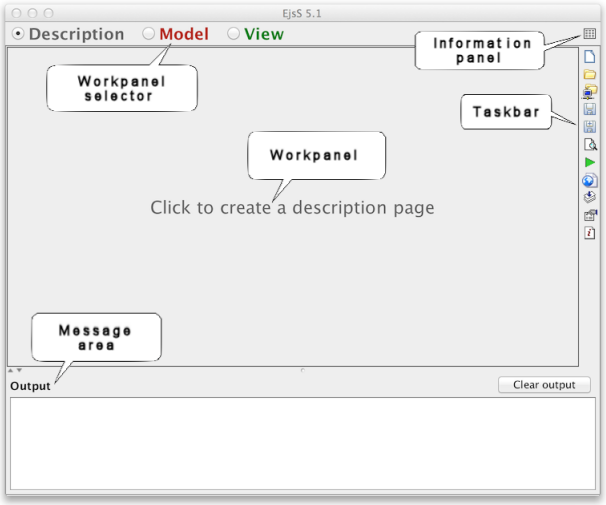

Figure 1.4: The *Easy Java/Javascript Simulations* user interface with annotations.

The taskbar on the right provides a series of icons to clear, open, search, and save a file, configure *EjsS*, and display program information and help. It also provides icons to run a simulation and to package one or more simulations in a self-contained file. Right-clicking on taskbar icons invokes alternative (but related) actions that will be described as needed. The bottom part of the interface contains an output area where *EjsS* displays informational messages. The central part of the interface contains the workpanels where the modeling is done.

Easy Java/Javascript Simulations provides three workpanels for modeling. The first panel, *Description*, allows us to create and edit multimedia HTML-based narrative that describes the model. Each narrative page appears in a tabbed panel within the workpanel and right-clicking on the tab allows the user to edit the narrative or to import additional narrative. The second work panel, *Model*, is dedicated to the modeling process. We use this panel to create variables that describe the model, to initialize these variables, and to write algorithms that describe how this model changes in time. The third workpanel, *View* or *HtmlView* (for HTML5-based interfaces), is dedicated to the task of building the graphical user interface, which allows users to control the simulation and to display its output. We build the interface by selecting elements from palettes and adding them to the view's *Tree of elements*. For example, the *Interface* palette contains buttons, sliders, and input fields and the *2D Drawables* palette contains elements to plot 2D data.

Easy Java/Javascript Simulations supports more than one program-

ming languages to implement the algorithms required for the modeling process. The interface of all of them is based on the same principles, and learning to use one of them, leads to understanding them all. We sometimes refer to the different interfaces of *Easy Java/Javascript Simulations* for the different programming languages, as the *flavors* of *EjsS*. Hence, we may refer to the *Java flavor* or the *Javascript flavor* of *EjsS*, for instance.

The *EjsS* console launches, by default, an instance of *EjsS* which supports the Java programming language (i.e. an instance of the Java flavor of *EjsS*). You can change the supported programming language or flavor (using the *Programming language* selector in the *Basic* tab of the *EjsS* console), and launch another instance of *EjsS* for this language clicking the *Launch* EjsS button. Figure 1.5 displays the graphical user interface for the creation of Javascript models in *Easy Java/Javascript Simulations*.

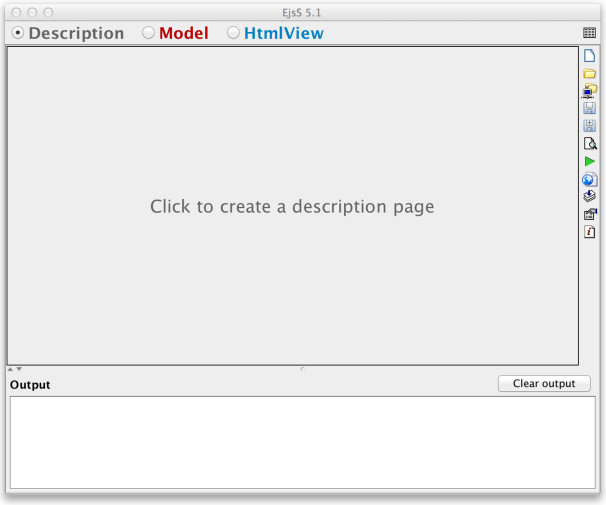

Figure 1.5: The *Easy Java/Javascript Simulations* user interface for Javascript support.

The next two chapters of this document guide you to inspect in detail and run an already existing simulation in each of the supported programming languages. (The simulation models the same physical phenomenon in all cases.) This will help you understand how the *Description*, *Model*, and *View* (or *HtmlView*) workpanels work together to help you model a simulation. If you are new to *Easy Java/Javascript Simulations*, please read the chapter for the *EjsS* flavour you are most interested in before exploring any of the many existing models in our digital libraries.

1.4 FINDING MODELS

Once you have covered the basics of *EjsS* and you know how to load, inspect, run, and even modify an example, you may be interested in finding existing simulations to see what other users have done with *EjsS*. Maybe, you can find a model that already fits your needs or that you can easily modify to be ready for classroom use.

There are two places you can look at to find more models. The first place to look at is the source sample directory that came with your distribution of *EjsS*. In the source directory of the distribution's workspace you will find some directories with sample simulations. These sample directories were also copied to your own workspace (unless you unselected this option) when you first run *EjsS*.

The second, and perhaps more interesting, place (actually places) to look for new models are available through the Internet. The *EjsS* digital libraries icon in the taskbar,  , opens a window which allows you to connect to repositories of *EjsS* models available through the Internet. This window, displayed in Figure 1.6, contains a combo box at its top that lists the available digital libraries. A second combo box lets you select the desired programming language for the models. Select one of these libraries, the programming language, or click the *Refresh* button to get the list of *EjsS* models in it. All these libraries work in a similar way, and we use the **comPADRE** digital library repository to illustrate how they are accessed from within *EjsS*.

Figure 1.6: The Digital Libraries window of *EjsS*. Select one of the available repositories using the combo box at the top of the window, the preferred programming language, or click the *Refresh* button to retrieve the list of models available.

The **comPADRE Pathway**, a part of the (USA) National Science Digital Library, is a growing network of educational resource collections

supporting teachers and students in Physics and Astronomy. Of special relevance for our interests is the Open Source Physics comPADRE collection available at <*http://www.compadre.org/OSP*>. This collection contains computational resources for teaching in the form of executable simulations and curriculum resources that engage students in physics, computation, and computer modeling. In particular, it contains *EjsS* models whose source (XML) code can be accessed directly from *EjsS* using the digital libraries icon.

If you are connected to the Internet, select the *OSP collection on the comPADRE digital library* entry of the top combo box and *EjsS* will connect to the library to obtain the very latest catalog of *EjsS* models in the library. At the moment of this writing, there are some 500 Java models and 200 Javascript models organized in different categories and subcategories, and the collection is expected to grow. As the left frame of Figure 1.7 shows, the collection is organized in categories and subcategories. When the name of a subcategory appears in red, double-click it to expand the node with the list of models of the subcategory. Because many models have primary and secondary classifications, a check box at the top pane, next to the *Search* field, allows you to decide whether you want the models to be listed uniquely under their primary classification, or appear in all matching categories (thus appearing more than once).

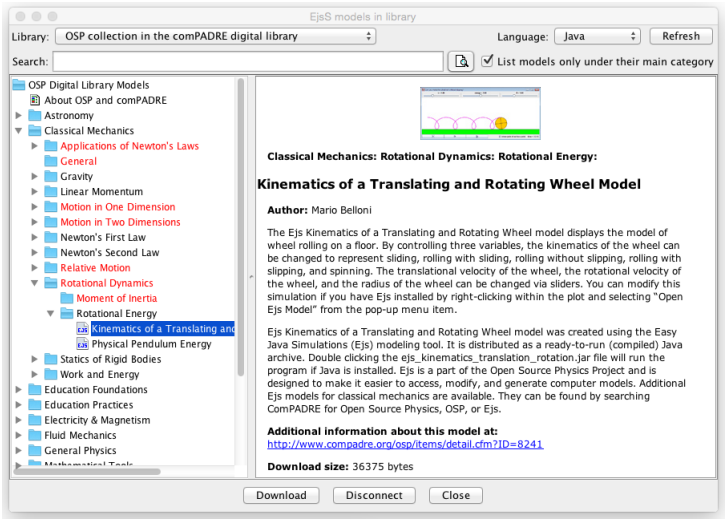

Figure 1.7: The OSP collection on the comPADRE digital library. The collection is organized in categories and subcategories. The entry for a model provides information about the model.

When you click a model node, the right frame shows information about the model obtained instantly from the library. The information describes the model, and includes a direct link to the comPADRE library for further

information. Double-clicking the model entry, or clicking the *Download* button, will retrieve the model and auxiliary files from the library, ask you for a place in your source directory of your workspace to download them, and open the model in *EjsS* when the download is complete. Because source files are usually small, the download takes place almost instantly. Now, you can inspect, run, or modify the model as shown in the exploration chapters below, for the mass and spring model.

The OSP collection on the comPADRE digital library is a highly recommended place to look for *EjsS* models and accompanying curricular material.

1.5 SUMMARY

Easy Java/Javascript Simulations is a modeling and authoring tool expressly devoted to the task of creating computer simulations to study and display a wide range of phenomena ranging from the simple to the complex. These computer simulations are used to obtain numerical data from our models as they advance in time, and to display this data in a form humans can understand.

EjsS has been designed to let us work at a high conceptual level, concentrating most of our time on the scientific aspects of our simulation, and asking the computer to automatically perform all the other necessary but easily automated tasks. Every tool, including *Easy Java/Javascript Simulations*, has a learning curve. The rest of this document contains a series of detailed examples that will familiarize you with the modeling, visualization, and interaction capabilities of *EjsS*.

Modeling is both a science and an art. *Easy Java/Javascript Simulations* is a tool that lets you express your science knowledge by facilitating the techniques required by the art, and by providing simple access to many modeling examples from other authors.

Chapter Two

Exploring the Java flavor of Easy Java/Javascript Simulations

A good example is the best sermon. *Benjamin Franklin*

To provide a perspective of the modeling process, in this chapter we first load, inspect, and run an existing simple harmonic oscillator simulation. We then modify the simulation to show how *EjsS* engages the user in the modeling process and greatly reduces the amount of programming that is required. This chapter uses Java as the programming language for the modeling and is a twin chapter of Chapter A (where Javascript is used).

2.1 INSPECTING THE SIMULATION

As mentioned in Chapter 1, *Easy Java/Javascript Simulations* provides three workpanels for modeling. The first panel, *Description*, allows us to create and edit multimedia HTML-based narrative that describes the model. The second work panel, *Model*, is dedicated to the modeling process. We use this panel to create variables that describe the model, to initialize these variables, and to write algorithms that describe how this model changes in time. The third workpanel, *View*, is dedicated to the task of building the graphical user interface, which allows users to control the simulation and to display its output.

To understand how the *Description*, *Model*, and *View* workpanels work together, we inspect and run an already existing simulation. Screen shots are no substitute for a live demonstration, and you are encouraged to follow along on your computer as you read.

Click on the *Open* icon on the *EjsS* taskbar. A file dialog similar to that in Figure 2.1 appears showing the contents of your workspace's **source** directory. Go to the **JavaExamples** directory, where you will find a file called **MassAndSpring.ejs**. Select this file and click on the *Open* button

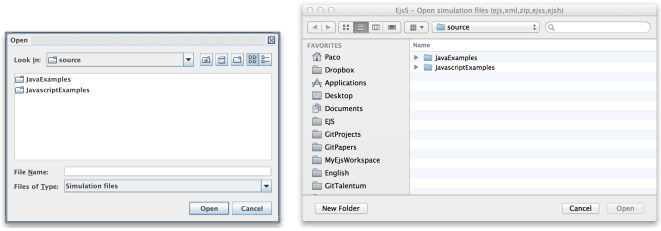

Figure 2.1: The open file dialog lets you browse your hard disk and load an existing simulation. The appearance of the dialog (shown here using two different look and feels) may vary, depending on your operating system and the selected *look and feel*.

of the file dialog.

Now, things come to life! *EjsS* reads the **MassAndSpring.ejs** document which populates the workpanels and two new "Ejs windows" appear in your display as shown in Figure 2.2. A quick warning. You can drag objects within these mock-up windows but this will set the model's initial conditions. It is usually better to set initial conditions using a table of variables as described in Section 2.1.2.

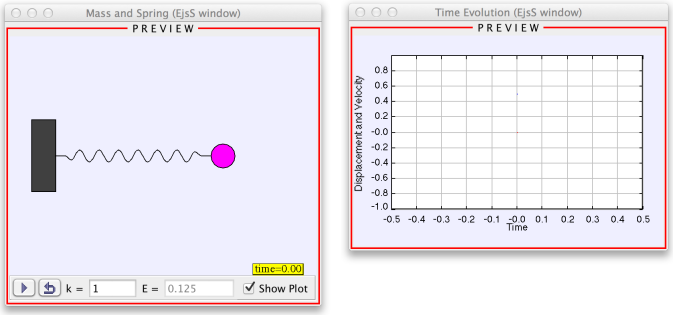

Figure 2.2: *EjsS* mock-up windows of the **MassAndSpring** simulation. The title bar and the red border show that they are *EjsS*' windows and that the program is not running.

Impatient or precocious readers may be tempted to click on the green run icon on the taskbar to execute our example before proceeding with this tutorial. Readers who do so will no longer be interacting with *EjsS* but with a compiled and running Java program. Exit the running program by closing the *Mass and Spring* window or by right clicking on the (now) red-squared stop icon on *EjsS*' taskbar before proceeding.

2.1.1 The *Description* workpanel

Select the *Description* workpanel by clicking on the corresponding radio button at the top of *EjsS*, and you will see two pages of narrative for this simulation. The first page, shown in Figure 2.3, contains a short discussion of the mass and spring model. Click on the *Activities* tab to view the second page of narrative.

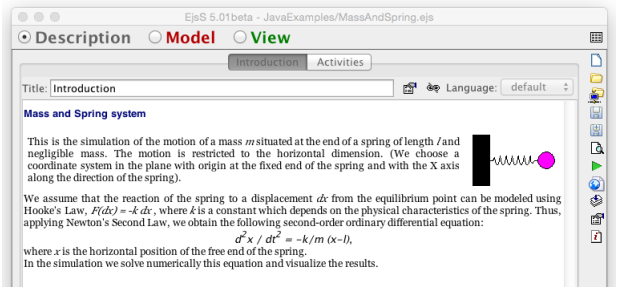

Figure 2.3: The description pages for the mass and spring simulation. Click on a tab to display the page. Right-click on a tab to edit the page.

A *Description* is HTML or XHTML multimedia text that provides information and instructions about the simulation. HTML stands for HyperText Markup Language and is the most commonly used protocol for formatting and displaying documents on the Web. The X in XHTML stands for eXtensible. XHTML is basically HTML expressed as valid XML or, in simpler words, perfectly formatted HTML.

EjsS provides a simple HTML editor that lets you create and modify pages within *EjsS*. You can also import HTML or (preferably) XHTML pages into *EjsS* by right clicking on a tab in the *Description* workpanel. (See Section 2.4.3.) Description pages are an essential part of the modeling process and these pages are included with the compiled model when the model is exported for distribution.

2.1.2 The *Model* workpanel

The *Model* workpanel is where the model is defined so that it can be converted into a program by *EjsS*. In this simulation, we study the motion of a particle of mass m attached to one end of a massless spring of equilibrium length L. The spring is fixed to the wall at its other end and is restricted to move in the horizontal direction. Although the oscillating mass has a well known analytic solution, it is useful to start with a simple harmonic oscillator model so that our output can be compared with an exact analytic

result.

Our model assumes small oscillations so that the spring responds to a given (horizontal) displacement δx from its equilibrium length L with a force given by Hooke's law, $F_x = -k\,\delta x$, where k is the elastic constant of the spring, which depends on its physical characteristics. We use Newton's second law to obtain a second-order differential equation for the position of the particle:

$$\frac{d^2\,x}{dt^2} = -\frac{k}{m}\,(x - L). \tag{2.1.1}$$

Notice that we use a system of coordinates with its x-axis along the spring and with its origin at the spring's fixed end. The particle is located at x and its displacement from equilibrium $\delta x = x - L$ is zero when $x = L$. We solve this system numerically to study how the state evolves in time.

Let's examine how we implement the mass and spring model by selecting the *Model* radio button and examining each of its six panels.

2.1.2.1 Declaration of variables

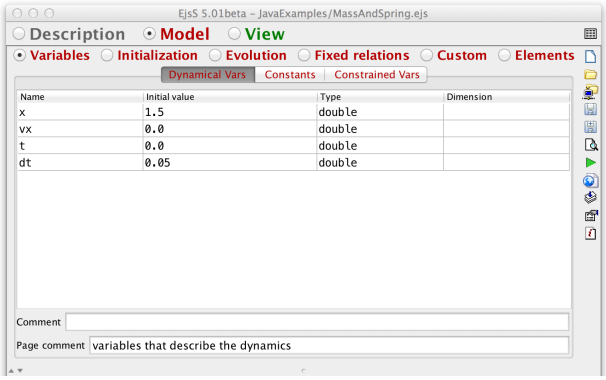

Figure 2.4: The *Model* workpanel contains six subpanels. The subpanel for the definition of mass and spring dynamical variables is displayed. Other tabs in this subpanel define additional variables, such as the natural length of the spring L and the energy E.

When implementing a model, a good first step is to identify, define, and initialize the variables that describe the system. The term *variable* is very general and refers to anything that can be given a name, including a physical constant and a graph. Figure 2.4 shows an *EjsS* variable table. Each row defines a variable of the model by specifying the name of the variable, its type, its dimension, and its initial value.

Variables in computer programs can be of several types depending on the data they hold. The most frequently used types are `boolean` for true/false values, `int` for integers, `double` for high-precision (≈ 16 significant digits) numbers, and `String` for text. We will use all these variable types in this document, but the mass and spring model uses only variables of type `double` and `boolean`.

Variables can be used as parameters, state variables, or inputs and outputs of the model. The tables in Figure 2.4 define the variables used within our model. We have declared a variable for the x-position of the particle, `x`, for its velocity in the x-direction, `vx`, for the time, `t`, and for the increment of time at each simulation step, `dt`. We define some variables, in this and other tabs, that do not appear in Equation(2.1.1). The reason for auxiliary variables, such as `vx` or the kinetic, potential, and total energies, will be made clear in what follows. The bottom part of the variables panel contains a comment field that provide a description of the role of each variable in the model. Clicking on a variable displays the corresponding comment.

2.1.2.2 Initialization of the model

Correctly setting initial conditions is important when implementing a model because the model must start in a physically realizable state. Our model is relatively simple, and we initialize it by entering values (or simple Java expressions such as `0.5*m*vx*vx`) in the *Initial value* column of the table of variables. *EjsS* uses these values when it initializes the simulation.

> Advanced models may require an initialization algorithm. For example, a molecular dynamics model may set particle velocities for an ensemble of particles. The *Initialization* panel allows us to define one or more pages of Java code that perform the required computation. *EjsS* converts this code into a Java method[1] and calls this method at start-up and whenever the simulation is reset. The mass and spring *Initialization* panel is not shown here because it is empty. See Subsection 2.1.2.4 for an example of how Java code appears in *EjsS*.

2.1.2.3 The evolution of the model

The *Evolution* panel allows us to write the Java code that determines how the mass and spring system evolves in time and we will use this option

[1] A Java method is similar to a function or a subroutine in procedural computer languages.

frequently for models not based on ordinary differential equations (ODEs). There is, however, a second option that allows us to enter ordinary differential equations, such as (2.1.1), without programming. *EjsS* provides a dedicated editor that lets us specify differential equations in a format that resembles mathematical notation and automatically generates the correct Java code.

Let's see how the differential equation editor works for the mass and spring model. Because ODE algorithms solve systems of first-order ordinary differential equations, a higher-order equation, such as (2.1.1), must be recast into a first-order system. We can do so by treating the velocity as an independent variable which obeys its own equation:

$$\frac{d\ x}{dt} = v_x \tag{2.1.2}$$

$$\frac{d\ v_x}{dt} = -\frac{k}{m}\,(x - L). \tag{2.1.3}$$

The need for an additional differential equation explains why we declared the `vx` variable in our table of variables.

Clicking on the *Evolution* panel displays the ODE editor shown in Figure 2.5. Notice that the ODE editor displays (2.1.2) and (2.1.3) (using

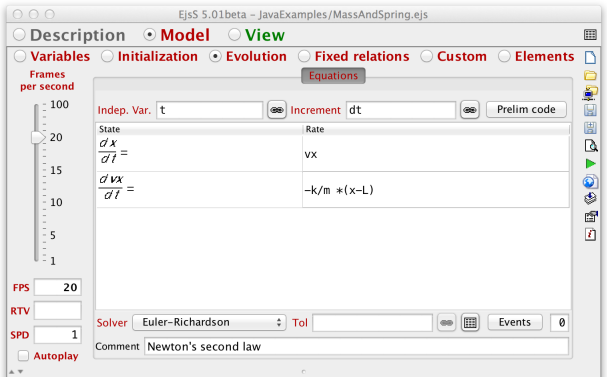

Figure 2.5: The ODE evolution panel showing the mass and spring differential equation and the numerical algorithm.

the * character to denote multiplication). Fields near the top of the editor specify the independent variable `t` and the variable increment `dt`. Numerical algorithms approximate the exact ODE solution by advancing the state in discrete steps and the increment determines this step size. The *Prelim code* button at the top-right of the editor allows us to enter preliminary code, to perform computations prior to evaluating the equations (a circumstance required in more complex situations than the one we treat in this example). A dropdown menu at the bottom of the editor lets us select the ODE solver

(numerical algorithm) that advances the solution from the current value of time, `t`, to the next value, `t + dt`. The tolerance field (*Tol*) is greyed out and empty because Euler–Richardson is a fixed-step method that requires no tolerance settings. The advanced button displays a dialog which allows us to fine-tune the execution of this solver, though default values are usually appropriated. Finally, the events field at the bottom of the panel tells us that we have not defined any events for this differential equation. Examples with preliminary code and events can be found further on in this document. The different solver algorithms and its parameters are discussed in the *EjsS* help.

The left-hand side of the evolution workpanel includes fields that determine how smoothly and how fast the simulation runs. The *frames per second* (*FPS*) option, which can be selected by using either a slider or an input field, specifies how many times per second we want our simulation to repaint the screen. The *steps per display* (*SPD*) input field specifies how many times we want to advance (step) the model before repainting. The current value of `20` frames per second produces a smooth animation that, together with the prescribed value of one step per display and `0.05` for `dt`, results in a simulation which runs at (approximately) real time. We will almost always use the default setting of one step per display. However, there are situations where the model's graphical output consumes a significant amount of processing power and where we want to speed the numerical computations. In this case we can increase the value of the steps per display parameter so that the model is advanced multiple times before the visualization is redrawn. The *Autoplay* check box indicates whether the simulation should start when the program begins. This box is unchecked so that we can change the initial conditions before starting the evolution.

The evolution workpanel handles the technical aspects of the mass and spring ODE model without programming. The simulation advances the state of the system by numerically solving the model's differential equations using the midpoint algorithm. The algorithm steps from the current state at time `t` to a new state at a new time `t + dt` before the visualization is redrawn. The simulation repeats this evolution step `20` times per second on computers with modest processing power. The simulation may run slower and not as smoothly on computers with insufficient processing power or if the computer is otherwise engaged, but it should not fail.

Although the mass and spring model can be solved with a simple ODE algorithm, our numerical methods library contains very sophisticated algorithms and *EjsS* can apply these algorithms to large systems of vector differential equations with or without discontinuous events.

2.1.2.4 Relations among variables

Not all variables within a model are computed using an algorithm on the Evolution workpanel. Variables can also be computed after the evolution has been applied. We refer to variables that are computed using the evolution algorithm as state variables or dynamical variables, and we refer to variables that depend on these variables as auxiliary or output variables. In the mass and spring model the kinetic, potential, and total energies of the system are output variables because they are computed from state variables.

$$T = \frac{1}{2} m {v_x}^2, \tag{2.1.4}$$

$$V = \frac{1}{2} k (x - L)^2, \tag{2.1.5}$$

$$E = T + V. \tag{2.1.6}$$

We say that there exists *fixed relations* among the model's variables.

The *Fixed relations* panel shown in Figure 2.6 is used to write relations among variables. Notice how easy it is to convert (2.1.4) through (2.1.6) into Java syntax. Be sure to use the multiplication character * and to place a semicolon at the end of each Java statement.

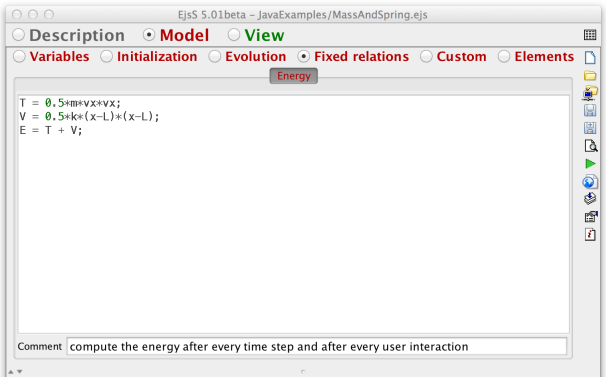

Figure 2.6: Fixed relations for the mass and spring model.

Here goes an important remark. You may wonder why we do not write fixed relation expressions by adding a second code page after the ODE page in the *Evolution* panel. After all, evolution pages execute sequentially and a second evolution page would correctly update the output variables after every step. The reason that the *Evolution* panel should not be used is that relations must *always* hold and there are other ways, such as mouse actions, to affect state variables. For example, dragging the mass changes the x variable and this change affects the energy. *EjsS* automatically evaluates the relations after initialization, after every evolution step, and whenever

there is any user interaction with the simulation's interface. For this reason, it is important that fixed relations among variables be written in the *Fixed relations* workpanel.

2.1.2.5 Custom pages

There is a fifth panel in the *Model* workpanel labeled *Custom*. This panel can be used to define Java methods (functions) that can be used throughout the model. This panel is empty because our model currently doesn't require additional methods, but we will make use of this panel when we modify our mass and spring example in Section 2.4. A custom method is not used unless it is explicitly invoked from another workpanel.

2.1.2.6 Model elements

The final, sixth panel in the *Model* workpanel is labeled *Elements* and provides access to third-party Java libraries in the form of drag and drop icons. You add these libraries to your program by dragging the corresponding icon to the list of model elements to use for this model. This creates Java objects you can then use in your model code. This panel is also empty for this model because our mass and spring doesn't require additional Java libraries.

2.1.3 The *View* workpanel

The third *Easy Java/Javascript Simulations* workpanel is the *View*. This workpanel allows us to create a graphical interface that includes visualization, user interaction, and program control with minimum programming. Figure 2.2 shows the view for the mass and spring model. Select the *View* radio button to examine how this view is created.

The right frame of the view workpanel of *EjsS*, shown in Figure 2.7, contains a collection of *view elements*, grouped by functionality. View elements are building blocks that can be combined to form a complete user interface, and each view element is a specialized object with an on-screen representation. To display information about a given element, click on its icon and press the *F1* key or right-click and select the *Help* menu item. To create a user interface, we create a frame (window) and add elements, such as buttons and graphs, using "drag and drop" as described in Section 2.4.

The *Tree of elements* shown on the left side of Figure 2.7 displays the

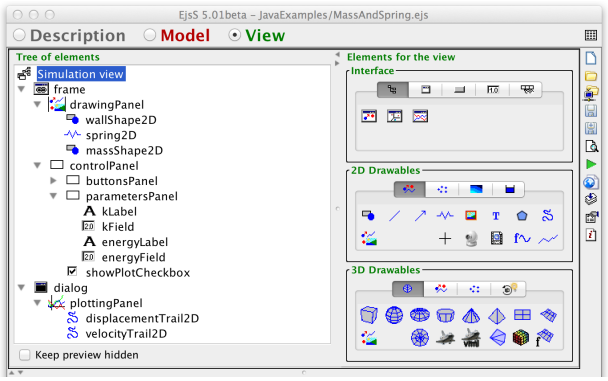

Figure 2.7: The *View* workpanel showing the *Tree of elements* for the mass and spring user interface.

structure of the mass and spring user interface. Notice that the simulation has two windows, a `Frame` and a `Dialog`, that appear on your computer screen. These elements belong to the class of *container* elements whose primary purpose is to visually group (organize) other elements within the user interface. The tree displays descriptive names and icons for these elements. Right-click on an element of the tree to obtain a menu that helps the user change this structure. Alternatively, you can drag and drop elements from one container to another to change the parent-child relationship, or within a container to change the child order. (There are conditions for a container to accept a given element as child. For instance, a two-dimensional drawing panel can only accept 2D drawable elements.)

Each view element has a set of internal parameters, called *properties*, which configure the element's appearance and behavior. We can edit these properties by double clicking on the element in the tree to display a table known as a *properties inspector*. Appearance properties, such as color, are often set to a constant value, such as `RED`. We can also use a variable from the model to set an element's property. This ability to connect (bind) a property to a variable without programming is the key to turning our view into a dynamic and interactive visualization.

Let's see how this procedure works in practice. Double-click on the `massShape2D` element (the 'Shape2D' suffix we added to the element's name helps you know the type of the element) in the tree to display the element's properties inspector. This element is the mass that is attached at the free end of the spring. The massShape2D's table of properties appears as shown in Figure 2.8.

Notice the properties that are given constant values. The `Style`, `Size X`, `Size Y`, and `Fill Color` properties produce an ellipse of size `(0.2,0.2)`

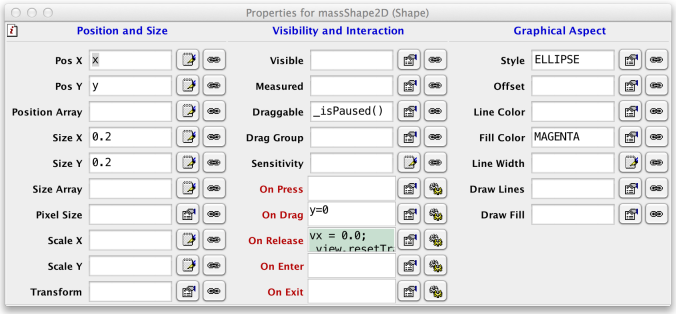

Figure 2.8: The table of properties of the `massShape2D` element.

units (which makes a circle) filled with the color magenta. More importantly, the `Pos X` and `Pos Y` properties of the shape are bound to the `x` and `y` variables of the model. This simple assignment establishes a bidirectional connection between model and view. These variables change as the model evolves and the shape follows the `x` and `y` values. If the user drags the shape to a new location, the `x` and `y` variables in the model change accordingly. Note that the `Draggable` property is only enabled when the animation is paused.

Elements can also have *action properties* which can be associated with code. (Action properties have their labels displayed in red.) User actions, such as dragging or clicking, invoke their corresponding action property, thus providing a simple way to control the simulation. As the user drags the mass, the code on the `On Drag` property restricts the motion of the shape to the horizontal direction by setting the `y` variable to `0`. Finally, when the mouse button is released, the following code is executed:

```
vx = 0.0;            // sets the velocity to zero
_view.resetTraces(); // clears all plots
```

Clicking on the icon next to the field displays a small editor that shows this code.

> Because the `On Release` action code spans more than one line, the property field in the inspector shows a darker (green) background. Other data types, such as boolean properties, have different editors. Clicking the second icon displays a dialog window with a listing of variables and methods that can be used to set the property value.

Exercise 2.1. Element inspectors

The mass' inspector displays different types of properties and their possible values. Explore the properties of other elements of the view. For instance,

the `displacementTrail2D` and `velocityTrail2D` elements correspond to the displacement and velocity time plots in the second window of the view, respectively. What is the maximum number of points that can be added to each trail? □

2.1.4 The completed simulation

We have seen that *Easy Java/Javascript Simulations* is a powerful tool that lets us express our knowledge of a model at a very high level of abstraction. When modeling the mass and spring, we first created a table of variables that describes the model and initialized these variables using a column in the table. We then used an evolution panel with a high-level editor for systems of first-order ordinary differential equations to specify how the state advances in time. We then wrote relations to compute the auxiliary or output variables that can be expressed using expressions involving state variables. Finally, the program's graphical user interface and high-level visualizations were created by dragging objects from the *Elements* palette into the *Tree of elements*. Element properties were set using a properties editor and some properties were associated with variables from the model.

It is important to note that the three lines of code on the Fixed relations workpanel (Figure 2.6) and the two lines of code in the particle's action method are the only explicit Java code needed to implement the model. *Easy Java/Javascript Simulations* creates a complete Java program by processing the information in the workpanels when the run icon is pressed as described in Section 2.2.

2.2 RUNNING THE SIMULATION

It is time to run the simulation by clicking on the *Run* icon of the taskbar, ▶. *EjsS* generates the Java code and compiles it, collects auxiliary and library files, and executes the compiled program. All at a single mouse click.

Running a simulation initializes its variables and executes the fixed relations to insure that the model is in a consistent state. The model's time evolution starts when the play/pause button in the user interface is pressed. (The play/pause button displays the ▶ icon when the simulation is paused and ⏸ when it is running.) In our current example, the program executes a numerical method to advance the harmonic oscillator differential equation by 0.05 time units and then executes the fixed relations code. Data are then passed to the graph and the graph is repainted. This process is repeated 20 times per second.

When running a simulation, *EjsS* changes its *Run* triangle icon to a red *Kill* square and prints informational messages saying that the simulation has been successfully generated and that it is running. Notice that the two *EjsS* windows disappear and are replaced by new but similar windows without the (Ejs window) suffix in their titles. These views respond to user actions. Click and drag the particle to a desired initial horizontal position and then click on the play/pause button. The particle oscillates about is equilibrium point and the plot displays the displacement and velocity data as shown in Figure 2.9.

Stop the simulation and right-click the mouse over any of the drawing areas of the simulation. In the popup menu that appears, select the `Elements options->plottingPanel->Data Tool` entry to display and analyze the data generated by the model. The same popup menu offers other run-time options, such as screen capture. To exit the program, close the simulation's main window.

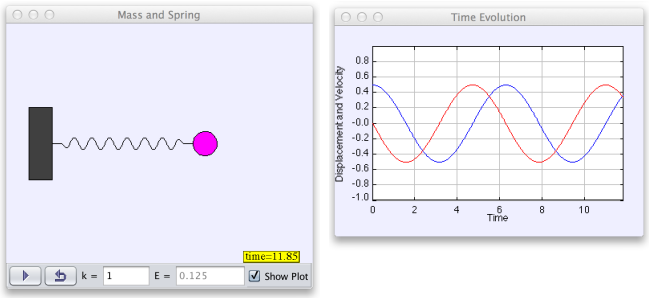

Figure 2.9: The mass and spring simulation displays an interactive drawing of the model and a graph with displacement and velocity data.

2.3 DISTRIBUTING THE SIMULATION

Simulations created with *EjsS* are stand-alone Java programs that can be distributed without *EjsS* for other people to use. The easiest way to do this is to package the simulation in a single executable jar file by clicking on the *Package* icon, . A file browser appears that lets you choose a name for the self-contained jar package. The default target directory to hold this package file is the **export** directory of your workspace, but you can choose any directory and package name. The stand-alone jar file is ready to be distributed on a CD or via the Internet. Other distribution mechanisms are available by right-clicking on the icon.

Exercise 2.2. Distribution of a model

Click on the *Package* icon on the taskbar to create a stand alone jar archive of the mass and spring simulation. Copy this jar file into a working direc-

tory separate from your *EjsS* installation. Close *EjsS* and verify that the simulation runs as a stand-alone application. □

Although the mass and spring jar file is a ready to use and to distribute Java application, an important pedagogic feature is that this jar file is created in such a way that users can return to *EjsS* at any time to examine, modify, and adapt the model. (*EjsS* must, of course, be installed.) The jar file contains a small *Extensible Markup Language* (XML) description of each model and right clicking on a drawing panel within the model brings in a popup menu with an option to copy this file into *EjsS*. This action will extract the required files from the jar, search for the *EjsS* installation in the user's hard disk, copy the files into the correct location, and run *EjsS* with this simulation loaded. If a model with the same name already exits, it can be replaced. The user can then inspect, run, and modify the model just as we are doing in this chapter. A student can, for example, obtain an example or a template from an instructor and can later repackage the modified model into a new jar file for submission as a completed exercise.

Exercise 2.3. Extracting a model
Run the stand-alone jar file containing the mass and spring model created in Exercise 2.2. Right click on the model's plot or drawing and select the *Open Ejs Model* item from the popup menu to copy the packaged model back into *EjsS*. □

EjsS is designed to be both a modeling and an authoring tool, and we suggest that you now experiment with it to learn how you can create and distribute your own models. As a start, we recommend that you run the mass and spring simulation and go through the activities in the second page of the *Description* workpanel. We modify this simulation in the next section.

2.4 MODIFYING THE SIMULATION

As we have seen, a prominent and distinctive feature of *Easy Java/Javascript Simulations* is that it allows us to create and study a simulation at a high level of abstraction. We inspected an existing mass and spring model and its user interface in the previous section. We now illustrate additional capabilities of *Easy Java/Javascript Simulations* by adding friction and a driving force and by adding a visualization of the system's phase space.

2.4.1 Extending the model

We can add damping in our model by introducing a viscous (Stoke's law) force that is proportional to the negative of the velocity $F_f = -b\,v_x$ where b is the damping coefficient. We also add an external time-dependent driving force which takes the form of a sinusoidal function $F_e(t) = A\,\sin(\omega\,t)$. The introduction of these two forces changes the second-order differential equation (2.1.1) to

$$\frac{d^2\,x}{dt^2} = -\frac{k}{m}\,(x - L) - \frac{b}{m}\,\frac{d\,x}{dt} + \frac{1}{m}\,F_e(t), \tag{2.4.1}$$

or, as in equations (2.1.2) and (2.1.3):

$$\frac{d\,x}{dt} = v_x, \tag{2.4.2}$$

$$\frac{d\,v_x}{dt} = -\frac{k}{m}\,(x - L) - \frac{b}{m}\,v_x + \frac{1}{m}\,F_e(t). \tag{2.4.3}$$

2.4.1.1 Adding variables

The introduction of new force terms requires that we add variables for the coefficient of dynamic friction and for the amplitude and frequency of the sinusoidal driving force. Return to the *Model* workpanel of *EjsS* and select its *Variables* panel. Right-click on the tab of the existing page of variables to see its popup menu, as in Figure 2.10. Select the *Add a new page* entry as shown in Figure 2.10. Enter `Damping and Driving Vars` for the new table name in the dialog and an empty table will appear.

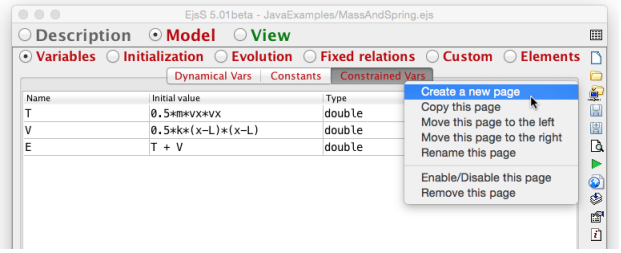

Figure 2.10: The popup menu for a page of variables.

We now use the new table to declare the needed variables. We could have used the already existing tables, but declaring multiple pages helps us organize the variables by category. Double-click on a table cell to make it editable and navigate through the table using the arrows or tab keys. Type `b` in the *Name* cell of the first row, and enter the value `0.1` in the *Initial value* cell to its right. We don't need to do anything else because the `double`

type selected is already correct. *EjsS* checks the syntax of the value entered and evaluates it. If we enter a wrong value, the background of the value cell will display a pink background. Notice that when you fill in a variable name, a new row appears automatically. Proceed similarly to declare a new variable for the driving force's `amp` with value `0.2` and for its `freq` with value `2.0`. Document the meaning of these variables by typing a short comment for each at the bottom of the table. Our final table of variables is shown in Figure 2.11. You can ignore the empty row at the end of the table or remove it by right-clicking on that row and selecting *Delete* from the popup menu that appears.

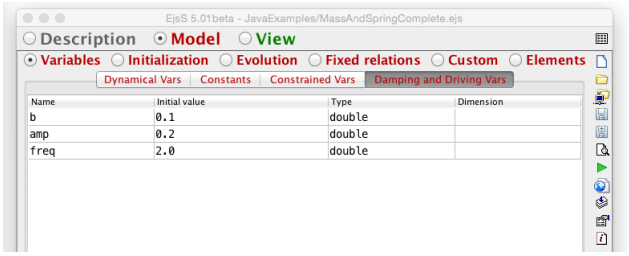

Figure 2.11: The new table of variables for the damping and forcing terms.

2.4.1.2 Modifying the evolution

We now modify the differential equations on the evolution page by adding expressions for the new terms in equation (2.4.3). Go to the evolution panel, double-click on the *Rate* cell of the second equation, and edit it to read:

```
-k/m * (x-L) - b*vx/m + force(t)/m
```

Notice that we are using a method (function) named `force` that has not yet been defined. We could have written an explicit expression for the sinusoidal function. However, defining a `force` method promotes cleaner and more readable code and allows us to introduce custom methods.

2.4.1.3 Adding custom code

The `force` method is defined using the *Custom* panel of the *Model*. Go to this panel and click on the empty central area to create a new page of custom code. Name this page *force*. You will notice that the page is created with a code template that defines the method. Edit this code to read:

```
public double force (double time) {
  return amp*Math.sin(freq*time); // sinusoidal driving force
}
```

Type this code exactly as shown including capitalization. Compilers complain if there is any syntax error.

Notice that we pass the time at which we want to compute the driving force to the `force` method as an input parameter. Passing the time value is very important. It would be incorrect to ask the method to use the value of the variable `t`, as in:

```
public double force () { // incorrect implementation of the force method
  return amp*Math.sin(freq*t);
}
```

The reason that time must be passed to the method is that time changes throughout the evolution step. In order for the ODE solver to correctly compute the time-dependent force throughout the evolution step, the time must be passed into the method that computes the rate.

> Variables that change (evolve) must be passed to methods that are used to compute the rate because numerical solvers evaluate the *Rate* column in the ODE workpanel at intermediate values between t and $t + dt$. In other words, the independent variable and any other dynamic variable which is differentiated in the *State* column of the ODE editor must be passed to any method that is called in the *Rate* column. Variables which remain constant during an evolution step may be used without being passed as input parameters because the value of the variable at the beginning of the evolution step can be used.

2.4.2 Improving the view

We now add a visualization of the phase space (displacement versus velocity) of the system's evolution to the *View*. We also add new input fields to display and modify the value of the damping, amplitude, and frequency parameters.

Go to the *View* workpanel and notice that the *Interface* palette contains many subpanels. Click on the tab with the icon to display the *Windows, containers, and drawing panels* palette of view elements. Click on the icon for a plotting panel, , in this palette. You can rest (hover) the mouse cursor over an icon to display a hint that describes the element if

you have difficulty recognizing the icon. Selecting an element sets a colored border around its icon on the palette and changes the cursor to a magic wand, . These changes indicate that *EjsS* is ready to create an element of the selected type. (Return to the design mode –get rid of the magic wand– by clicking on any blank area within the *Tree of elements* or hitting the *Esc* key.)

Click on the `dialog` element in the *Tree of elements* as shown in Figure 2.12 to add the plotting panel to the view.

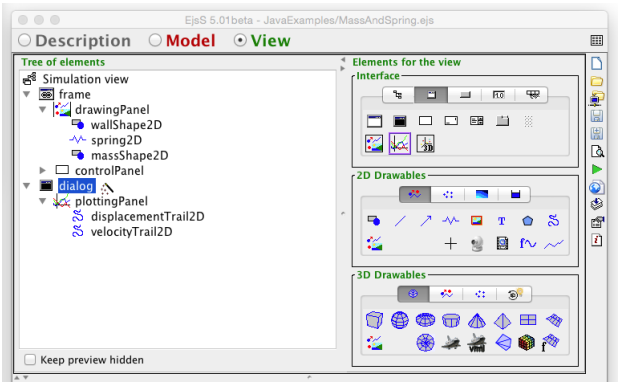

Figure 2.12: Creation of a plotting panel as a child of the `dialog` element of the view.

EjsS asks for the name of the new element and then creates the element as a child within the existing `dialog`. A new plot appears but the dialog is too small. Resize the dialog box by dragging its corner. You can also resize the dialog box by double-clicking on the `dialog` element in the tree to show its properties table and changing its `Size` property to `"385,524"`, thus doubling its height. Finally, edit the properties table of the newly created plotting panel element to set the `Title` property to `Phase Space`, the `Title X` property to `Displacement`, and the `Title Y` property to `Velocity`. (*EjsS* will add leading and trailing quotes to these strings to conform to the correct Java syntax.) Set the minima and maxima for both X and Y scales to `-1` and 1, respectively, and leave the other properties untouched.

The plotting panel is, as its name suggests, the container for the phase-space plot. Phase space data are drawn in this panel using an element of type `Trail2D`, . Find the `Trail2D` element in the `2D Drawables` palette and follow the same procedure as before. Select the `Trail2D` element and create an element of this type by clicking with the magic wand on the phase space panel. Finally, edit the properties of the new trail element to set its `Input X` property to `x - L` and its `Input Y` property to `vx`. This assignment causes the simulation to add a new `(x - L,vx)` point to the trace after each

evolution step, thus drawing the phase-space plot shown in Figure 2.13.

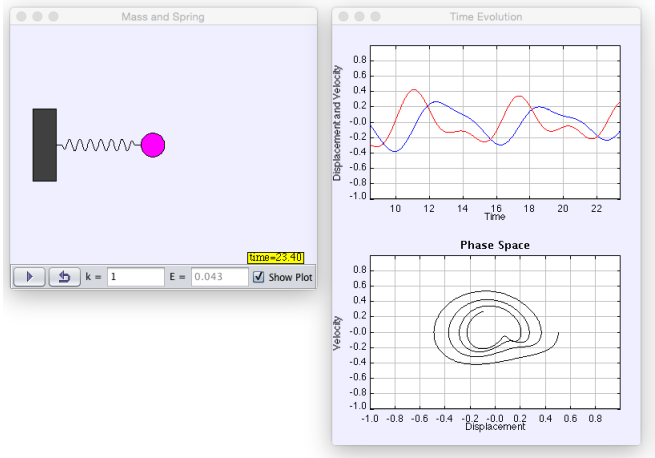

Figure 2.13: The modified simulation. The dialog includes now both a time and a phase-space plot.

To finish the modifications, we add a new panel to the top of the drawing frame that shows the sinusoidal driving force parameters.

- Select the `Panel` element icon, ▭, on the *Windows, containers, and drawing panels* subgroup of the *Interface* palette. Click with the magic wand on the element named `frame` within the *Tree of elements* to create a new panel named `forceParamPanel` in the frame's top location. Use the properties inspector to set this panel's layout to *FLOW:center,0,0* and its border type to `LOWERED_ETCHED`.
- Select the `Label` element icon, **A**, on the *Buttons and decorations* subgroup of the *Interface* palette and create a new element of that type in the `forceParamPanel` panel. Set the label's text property to `"frequency ="`.
- Select the `Field` element icon, 2.0, and create a new element named `freqField` in the force parameter panel. Edit the `freqField` properties table as shown in Figure 2.14. The connection to the `freq` variable is established using the `Variable` property. Click on the second icon to the right of the property field, 🔗, and choose the appropriate variable. The variable list shows all the model variables that can be used to set the property field. The `Format` property indicates the number of decimal digits with which to display the value of the variable.
- Repeat this process to add the `amp` variable to the user interface.

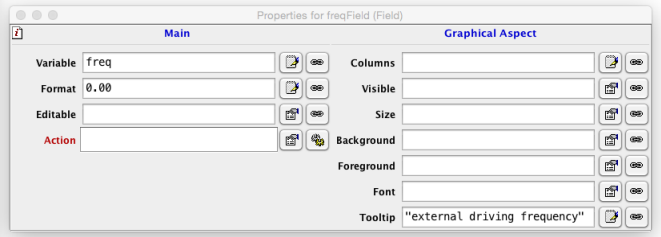

Figure 2.14: The table of properties of the `freqField` element.

2.4.3 Changing the description

Now that we have changed the model and the view, we should modify the description pages of our simulation. Go to the *Description* workpanel and click on the tab of the first page, the one labeled `Introduction`. Once you see this page, click the *Click to modify the page* icon, . The description page will change to edit mode, as shown in Figure 2.15, and a simple editor will appear that provides direct access to common HTML features.

If you prefer to use your own editor, you can copy and paste HTML fragments from your editor into the *EjsS* editor. If you know HTML syntax, you can enter tagged (markup) text directly by clicking the source icon, <>, in the tool bar. You can even import entire HTML pages into *EjsS* by clicking the *Link/Unlink page to external file* icons, , .

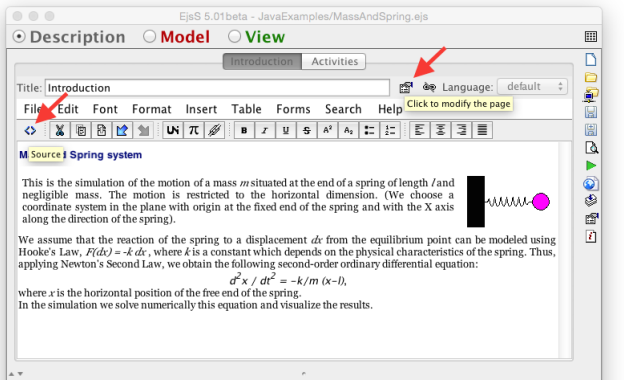

Figure 2.15: The HTML editor of *EjsS*. The added red arrows point to the edit and source code edition mode icons.

Edit the description pages as you find convenient. At least change the discussion of the model to include the damping and driving forces. When you are done, save the new simulation with a different name by clicking the *Save as* icon of *EjsS*' taskbar, . When prompted, enter a new name for your simulation's XML file. The modified simulation is stored in the **Mas-**

sAndSpringComplete.ejs file in the **source** directory for this chapter.

2.5 PROBLEMS AND PROJECTS

Problem 2.1 (Energy). Add a third plotting panel to the dialog window of the **MassAndSpringComplete.ejs** simulation that will display the evolution of the kinetic, potential, and total energies.

Problem 2.2 (Function plotter). The analytic solution for the undriven simple harmonic oscillator is

$$x(t) = A\sin(w_0 t + \phi) \tag{2.5.1}$$

where A is the amplitude (maximum displacement), $w_0 = \sqrt{k/m}$ is the natural frequency of oscillation, and ϕ is the phase angle. Consult a mechanics textbook to determine the relationship between the amplitude and phase angle and the initial displacement and velocity. Use the **FunctionPlotter.ejs** simulation in the examples directory to compare the analytic solution to the numerical solution generated by the **MassAndSpringComplete.ejs** model.

Project 2.1 (Two-dimensional oscillator). Modify the model of the mass and spring simulation to consider motion that is not restricted to the horizontal direction. Assume that a second spring with spring constant k' produces a vertical restoring force $F_y(\delta y) = -k'\,\delta y$. Modify the simulation to allow the user to specify the Hooke's law constants as well as the initial conditions in both directions. Describe the motion produced without a driving force but under different initial conditions and with different spring constants. (Try $k = 1$ and $k' = 9$.) Show that it is possible to obtain circular motion if $k = k'$.

Project 2.2 (Simple pendulum). Create a similar simulation as the one described in this chapter for a simple pendulum whose second-order differential equation of motion is

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin(\theta), \tag{2.5.2}$$

where θ is the angle of the pendulum with the vertical, g is the acceleration due to gravity, and L is the arms's length. Use fixed relations to compute the x and y position of the pendulum bob using the equations:

$$x = L\,\sin(\theta)$$
$$y = -L\,\cos(\theta).$$

Chapter Three

Exploring the Javascript flavor of Easy Java/Javascript Simulations

> Change is the law of life. And those who look only to the past or present are certain to miss the future. *John F. Kennedy*

To provide a perspective of the modeling process, in this chapter we first load, inspect, and run an existing simple harmonic oscillator simulation. We then modify the simulation to show how *EjsS* engages the user in the modeling process and greatly reduces the amount of programming that is required. This chapter uses Javascript as the programming language for the modeling and is a twin chapter for Chapter 2 (where Java is used).

3.1 INSPECTING THE SIMULATION

As mentioned in Chapter 1, *Easy Java/Javascript Simulations* provides three workpanels for modeling. The first panel, *Description*, allows us to create and edit multimedia HTML-based narrative that describes the model. The second work panel, *Model*, is dedicated to the modeling process. We use this panel to create variables that describe the model, to initialize these variables, and to write algorithms that describe how this model changes in time. The third workpanel, *HtmlView*, is dedicated to the task of building the graphical user interface, which allows users to control the simulation and to display its output.

To understand how the *Description*, *Model*, and *HtmlView* workpanels work together, we inspect and run an already existing simulation. Screen shots are no substitute for a live demonstration, and you are encouraged to follow along on your computer as you read.

Click on the *Open* icon on the *EjsS* taskbar. A file dialog similar to that in Figure 3.1 appears showing the contents of your workspace's **source** directory. Go to the **JavascriptExamples** directory, where you will find

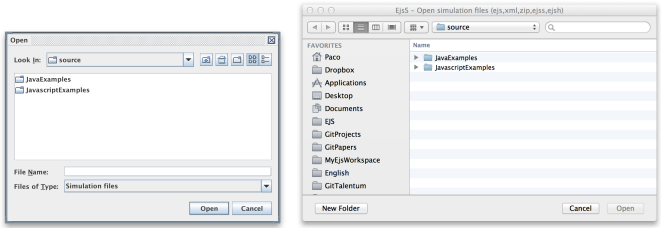

Figure 3.1: The open file dialog lets you browse your hard disk and load an existing simulation. The appearance of the dialog (shown here using two different look and feels) may vary, depending on your operating system and the selected *look and feel*.

a file called **MassAndSpring.ejss**. Select this file and click on the *Open* button of the file dialog.

Now, things come to life! *EjsS* reads the **MassAndSpring.ejss** document which populates the workpanels and a new "EjsS Emulator" appears in your display as shown in Figure 3.2. A quick warning. You can drag objects or click buttons within this mock-up window but the model will not exhibit its full behavior. You need to run the simulation for that.

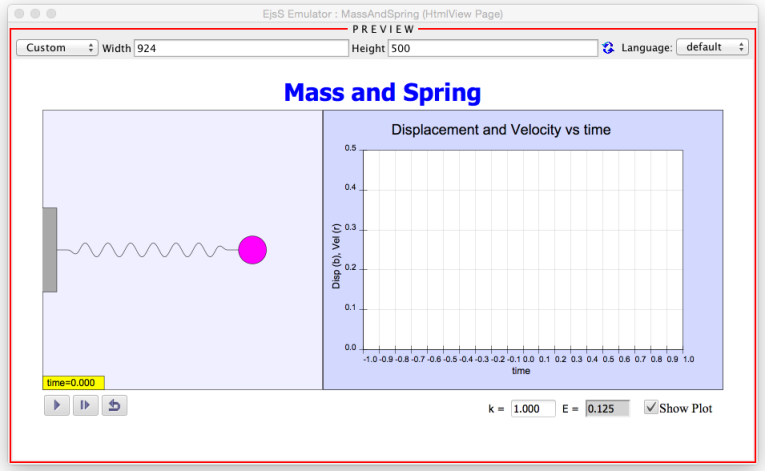

Figure 3.2: *EjsS* mock-up windows of the **MassAndSpring** simulation. The title bar and the red border show that this an HTML Emulator window within *EjsS* and that the program is not running.

Impatient or precocious readers may be tempted to click on the green run icon on the taskbar to execute our example before proceeding with this tutorial. Readers who do so will no longer be interacting with *EjsS* but with a compiled and running Javascript program on an HTML page. Exit the running program by closing the *Mass and Spring* window or by right clicking on the (now) red-squared stop icon on *EjsS*' taskbar before proceeding.

3.1.1 The *Description* workpanel

Select the *Description* workpanel by clicking on the corresponding radio button at the top of *EjsS*, and you will see two pages of narrative for this simulation. The first page, shown in Figure 3.3, contains a short discussion of the mass and spring model. Click on the *Activities* tab to view the second page of narrative.

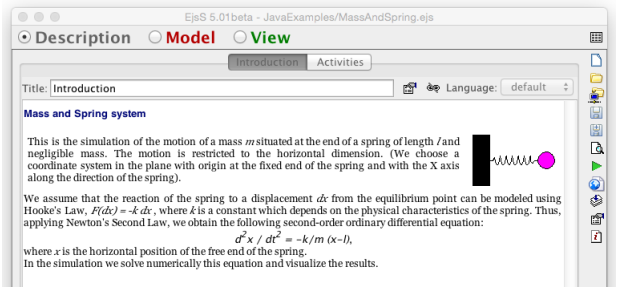

Figure 3.3: The description pages for the mass and spring simulation. Click on a tab to display the page. Right-click on a tab to edit the page.

A *Description* is HTML or XHTML multimedia text that provides information and instructions about the simulation. HTML stands for HyperText Markup Language and is the most commonly used protocol for formatting and displaying documents on the Web. The X in XHTML stands for eXtensible. XHTML is basically HTML expressed as valid XML or, in simpler words, perfectly formatted HTML.

EjsS provides a simple HTML editor that lets you create and modify pages within *EjsS*. You can also import HTML or (preferably) XHTML pages into *EjsS* by right clicking on a tab in the *Description* workpanel. (See Section 3.4.3.) Description pages are an essential part of the modeling process and these pages are included with the compiled model when the model is exported for distribution.

3.1.2 The *Model* workpanel

The *Model* workpanel is where the model is defined so that it can be converted into a program by *EjsS*. In this simulation, we study the motion of a particle of mass m attached to one end of a massless spring of equilibrium length L. The spring is fixed to the wall at its other end and is restricted to move in the horizontal direction. Although the oscillating mass has a well known analytic solution, it is useful to start with a simple harmonic oscillator model so that our output can be compared with an exact analytic

result.

Our model assumes small oscillations so that the spring responds to a given (horizontal) displacement δx from its equilibrium length L with a force given by Hooke's law, $F_x = -k\,\delta x$, where k is the elastic constant of the spring, which depends on its physical characteristics. We use Newton's second law to obtain a second-order differential equation for the position of the particle:

$$\frac{d^2\,x}{dt^2} = -\frac{k}{m}\,(x - L). \tag{3.1.1}$$

Notice that we use a system of coordinates with its x-axis along the spring and with its origin at the spring's fixed end. The particle is located at x and its displacement from equilibrium $\delta x = x - L$ is zero when $x = L$. We solve this system numerically to study how the state evolves in time.

Let's examine how we implement the mass and spring model by selecting the *Model* radio button and examining each of its six panels.

3.1.2.1 Declaration of variables

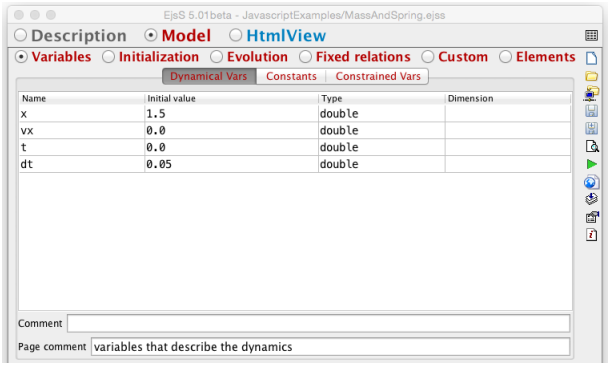

Figure 3.4: The *Model* workpanel contains six subpanels. The subpanel for the definition of mass and spring dynamical variables is displayed. Other tabs in this subpanel define additional variables, such as the natural length of the spring L and the energy E.

When implementing a model, a good first step is to identify, define, and initialize the variables that describe the system. The term *variable* is very general and refers to anything that can be given a name, including a physical constant and a graph. Figure 3.4 shows an *EjsS* variable table. Each row defines a variable of the model by specifying the name of the variable, its type, its dimension, and its initial value.

Variables in computer programs can be of several types depending

on the data they hold. The most frequently used types are `boolean` for true/false values, `int` for integers, `double` for high-precision (≈ 16 significant digits) numbers, and `String` for text. We will use all these variable types in this document, but the mass and spring model uses only variables of type `double` and `boolean`.

> If you have already learnt a bit of Javascript, you will probably know that Javascript has actually no types for variables. All variables are declared with a *var* keyword. This means that, in principle, Javascript makes no difference among integers, doubles, Strings, etc... But it does! For instance, you should not use a double variable as index for an array. For this reason, and also because it helps clarify the use of variables in your model (what values they can have and where you can or cannot use them), we ask you to assign a type to each variable in your model.

Variables can be used as parameters, state variables, or inputs and outputs of the model. The tables in Figure 3.4 define the variables used within our model. We have declared a variable for the x-position of the particle, `x`, for its velocity in the x-direction, `vx`, for the time, `t`, and for the increment of time at each simulation step, `dt`. We define some variables, in this and other tabs, that do not appear in Equation(3.1.1). The reason for auxiliary variables, such as `vx` or the kinetic, potential, and total energies, will be made clear in what follows. The bottom part of the variables panel contains a comment field that provide a description of the role of each variable in the model. Clicking on a variable displays the corresponding comment.

3.1.2.2 Initialization of the model

Correctly setting initial conditions is important when implementing a model because the model must start in a physically realizable state. Our model is relatively simple, and we initialize it by entering values (or simple Javascript expressions such as `0.5*m*vx*vx`) in the *Initial value* column of the table of variables. *EjsS* uses these values when it initializes the simulation.

> Advanced models may require an initialization algorithm. For example, a molecular dynamics model may set particle velocities for an ensemble of particles. The *Initialization* panel allows us to define one or more pages of Javascript code that perform the required computation. *EjsS* converts this code into a Javascript function and calls this method at start-up and whenever the simulation is reset. The mass and spring *Initialization* panel is not shown here because it is empty. See Subsection 3.1.2.4 for an example of how Javascript code appears in *EjsS*.

3.1.2.3 The evolution of the model

The *Evolution* panel allows us to write the Javascript code that determines how the mass and spring system evolves in time and we will use this option frequently for models not based on ordinary differential equations (ODEs). There is, however, a second option that allows us to enter ordinary differential equations, such as (3.1.1), without programming. *EjsS* provides a dedicated editor that lets us specify differential equations in a format that resembles mathematical notation and automatically generates the correct Javascript code.

Let's see how the differential equation editor works for the mass and spring model. Because ODE algorithms solve systems of first-order ordinary differential equations, a higher-order equation, such as (3.1.1), must be recast into a first-order system. We can do so by treating the velocity as an independent variable which obeys its own equation:

$$\frac{d\ x}{dt} = v_x \tag{3.1.2}$$

$$\frac{d\ v_x}{dt} = -\frac{k}{m}\,(x - L). \tag{3.1.3}$$

The need for an additional differential equation explains why we declared the `vx` variable in our table of variables.

Clicking on the *Evolution* panel displays the ODE editor shown in Figure 3.5. Notice that the ODE editor displays (3.1.2) and (3.1.3) (using

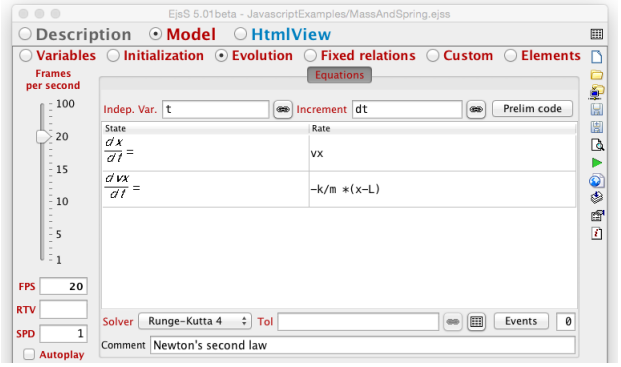

Figure 3.5: The ODE evolution panel showing the mass and spring differential equation and the numerical algorithm.

the * character to denote multiplication). Fields near the top of the editor specify the independent variable `t` and the variable increment `dt`. Numerical algorithms approximate the exact ODE solution by advancing the state in discrete steps and the increment determines this step size. The *Prelim code*

button at the top-right of the editor allows us to enter preliminary code, to perform computations prior to evaluating the equations (a circumstance required in more complex situations than the one we treat in this example). A dropdown menu at the bottom of the editor lets us select the ODE solver (numerical algorithm) that advances the solution from the current value of time, `t`, to the next value, `t + dt`. The tolerance field (*Tol*) is greyed out and empty because Runge–Kutta 4 is a fixed-step method that requires no tolerance settings. The advanced button displays a dialog which allows us to fine-tune the execution of this solver, though default values are usually appropriated. Finally, the events field at the bottom of the panel tells us that we have not defined any events for this differential equation. Examples with preliminary code and events can be found further on in this document. The different solver algorithms and its parameters are discussed in the *EjsS* help.

The left-hand side of the evolution workpanel includes fields that determine how smoothly and how fast the simulation runs. The *frames per second* (*FPS*) option, which can be selected by using either a slider or an input field, specifies how many times per second we want our simulation to repaint the screen. The *steps per display* (*SPD*) input field specifies how many times we want to advance (step) the model before repainting. The current value of `20` frames per second produces a smooth animation that, together with the prescribed value of one step per display and `0.05` for `dt`, results in a simulation which runs at (approximately) real time. We will almost always use the default setting of one step per display. However, there are situations where the model's graphical output consumes a significant amount of processing power and where we want to speed the numerical computations. In this case we can increase the value of the steps per display parameter so that the model is advanced multiple times before the visualization is redrawn. The *Autoplay* check box indicates whether the simulation should start when the program begins. This box is unchecked so that we can change the initial conditions before starting the evolution.

The evolution workpanel handles the technical aspects of the mass and spring ODE model without programming. The simulation advances the state of the system by numerically solving the model's differential equations using the midpoint algorithm. The algorithm steps from the current state at time `t` to a new state at a new time `t + dt` before the visualization is redrawn. The simulation repeats this evolution step `20` times per second on computers or devices with modest processing power. The simulation may run slower and not as smoothly on computers or devices with insufficient processing power or if the computer is otherwise engaged, but it should not fail.

Although the mass and spring model can be solved with a simple ODE algorithm, our numerical methods library contains very sophisticated algorithms and *EjsS* can apply these algorithms to large systems of vector differential equations with or without discontinuous events.

3.1.2.4 Relations among variables

Not all variables within a model are computed using an algorithm on the Evolution workpanel. Variables can also be computed after the evolution has been applied. We refer to variables that are computed using the evolution algorithm as state variables or dynamical variables, and we refer to variables that depend on these variables as auxiliary or output variables. In the mass and spring model the kinetic, potential, and total energies of the system are output variables because they are computed from state variables.

$$T = \frac{1}{2}m{v_x}^2, \tag{3.1.4}$$

$$V = \frac{1}{2}k(x-L)^2, \tag{3.1.5}$$

$$E = T + V. \tag{3.1.6}$$

We say that there exists *fixed relations* among the model's variables.

The *Fixed relations* panel shown in Figure 3.6 is used to write relations among variables. Notice how easy it is to convert (3.1.4) through (3.1.6) into Javascript syntax. Be sure to use the multiplication character * and to place a semicolon at the end of each Javascript statement.

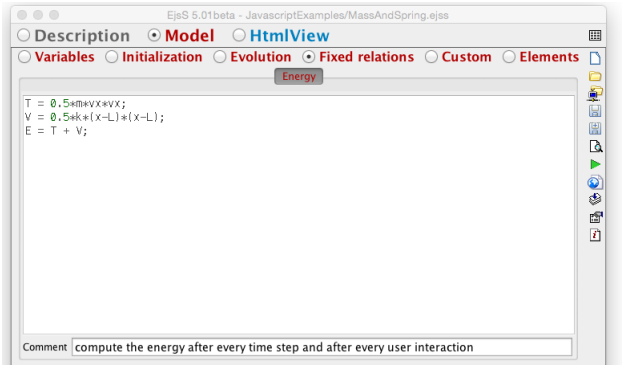

Figure 3.6: Fixed relations for the mass and spring model.

Here goes an important remark. You may wonder why we do not write fixed relation expressions by adding a second code page after the ODE page in the *Evolution* panel. After all, evolution pages execute sequentially and a second evolution page would correctly update the output variables after

every step. The reason that the *Evolution* panel should not be used is that relations must *always* hold and there are other ways, such as mouse actions, to affect state variables. For example, dragging the mass changes the x variable and this change affects the energy. *EjsS* automatically evaluates the relations after initialization, after every evolution step, and whenever there is any user interaction with the simulation's interface. For this reason, it is important that fixed relations among variables be written in the *Fixed relations* workpanel.

3.1.2.5 Custom pages

There is a fifth panel in the *Model* workpanel labeled *Custom*. This panel can be used to define Javascript functions that can be used throughout the model. This panel is empty because our model currently doesn't require additional methods, but we will make use of this panel when we modify our mass and spring example in Section 3.4. A custom method is not used unless it is explicitly invoked from another workpanel.

3.1.2.6 Model elements

The final, sixth panel in the *Model* workpanel is labeled *Elements* and provides access to third-party Javascript libraries in the form of drag and drop icons. You add these libraries to your program by dragging the corresponding icon to the list of model elements to use for this model. This creates Javascript objects you can then use in your model code. This panel is also empty for this model because our mass and spring doesn't require additional Javascript libraries.

3.1.3 The *HtmlView* workpanel

The third *Easy Java/Javascript Simulations* workpanel is the *HtmlView*. This workpanel allows us to create a graphical HTML-based interface that includes visualization, user interaction, and program control with minimum programming. Figure 3.2 shows the HTML-based view for the mass and spring model. Select the *HtmlView* radio button to examine how this HTML-based view is created.

The right frame of the *HtmlView* workpanel of *EjsS*, shown in Figure 3.7, contains a collection of HTML-based *view elements*, grouped by

functionality. View elements are building blocks that can be combined to form a complete user HTML-based interface, and each view element is a specialized object with an on-screen representation. To display information about a given element, click on its icon and press the *F1* key or right-click and select the *Help* menu item. To create a user interface, we create an empty HTML view (think of it as a blank HTML page) and add elements, such as panel, buttons and graphs, using "drag and drop" as described in Section 3.4.

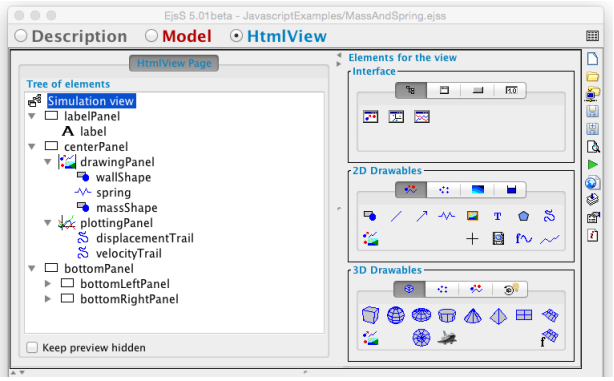

Figure 3.7: The *HtmlView* workpanel showing the *Tree of elements* for the mass and spring user interface.

The *Tree of elements* shown on the left side of Figure 3.7 displays the structure of the mass and spring user interface. Notice that the simulation has three main panels, `labelPanel`, `centerPanel` and `bottomPanel`, that appear tiled vertically on the HTML page in the Emulator. (The *EjsS* Emulator emulates an HTML browser.) These panels belong to the class of *container* elements whose primary purpose is to visually group (organize) other elements within the user interface. The tree displays descriptive names and icons for these elements. Right-click on an element of the tree to obtain a menu that helps the user change this structure. Alternatively, you can drag and drop elements from one container to another to change the parent-child relationship, or within a container to change the child order. (There are conditions for a container to accept a given element as child. For instance, a two-dimensional drawing panel can only accept 2D drawable elements.)

Each view element has a set of internal parameters, called *properties*, which configure the element's appearance and behavior. We can edit these properties by double clicking on the element in the tree to display a table known as a *properties inspector*. Appearance properties, such as color, are often set to a constant value, such as `"Red"`. We can also use a variable from the model to set an element's property. This ability to connect (bind) a property to a variable without programming is the key to turning our view

into a dynamic and interactive visualization.

Let's see how this procedure works in practice. Double-click on the `massShape` element (the 'Shape' suffix we added to the element's name helps you know the type of the element) in the tree to display the element's properties inspector. This element is the mass that is attached at the free end of the spring. The massShape's table of properties appears as shown in Figure 3.8.

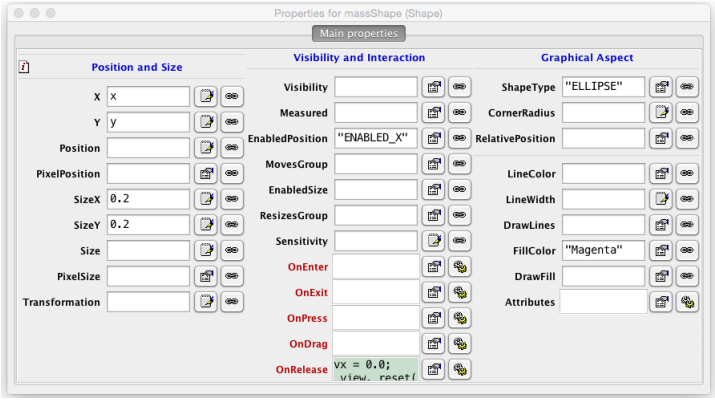

Figure 3.8: The table of properties of the `massShape` element.

Notice the properties that are given constant values. The `ShapeType`, `SizeX`, `SizeY`, and `FillColor` properties produce an ellipse of size `(0.2,0.2)` units (which makes a circle) filled with the color Magenta. More importantly, the `X` and `Y` properties of the shape are bound to the `x` and `y` variables of the model. This simple assignment establishes a bidirectional connection between model and view. These variables change as the model evolves and the shape follows the `x` and `y` values. If the user drags the shape to a new location, the `x` and `y` variables in the model change accordingly. Note, however, that the `Draggable` property is set to allow you only to drag horizontally the mass.

Elements can also have *action properties* which can be associated with code. (Action properties have their labels displayed in red.) User actions, such as dragging or clicking, invoke their corresponding action property, thus providing a simple way to control the simulation. When the user releases the mass after dragging it, the following code (specified on the `OnRelease` action property) is executed:

```
vx = 0.0;            // sets the velocity to zero
_view.reset(); // clears all plots
```

Clicking on the icon next to the field displays a small editor that shows this code.

> Because the `On Release` action code spans more than one line, the property field in the inspector shows a darker (green) background. Other data types, such as boolean properties, have different editors. Clicking the second icon displays a dialog window with a listing of variables and methods that can be used to set the property value.

Exercise 3.1. Element inspectors
The mass' inspector displays different types of properties and their possible values. Explore the properties of other elements of the view. For instance, the `displacementTrail` and `velocityTrail` elements correspond to the displacement and velocity time plots in the rightmost big plotting panel of the view, respectively. What is the maximum number of points that can be added to each trail? □

3.1.4 The completed simulation

We have seen that *Easy Java/Javascript Simulations* is a powerful tool that lets us express our knowledge of a model at a very high level of abstraction. When modeling the mass and spring, we first created a table of variables that describes the model and initialized these variables using a column in the table. We then used an evolution panel with a high-level editor for systems of first-order ordinary differential equations to specify how the state advances in time. We then wrote relations to compute the auxiliary or output variables that can be expressed using expressions involving state variables. Finally, the program's graphical user interface and high-level visualizations were created by dragging objects from the *Elements* palette into the *Tree of elements*. Element properties were set using a properties editor and some properties were associated with variables from the model.

It is important to note that the three lines of code on the Fixed relations workpanel (Figure 3.6) and the two lines of code in the particle's action method are the only explicit Javascript code needed to implement the model. *Easy Java/Javascript Simulations* creates a complete Javascript program by processing the information in the workpanels when the run icon is pressed as described in Section 3.2.

3.2 RUNNING THE SIMULATION

It is time to run the simulation by clicking on the *Run* icon of the taskbar, ▶. *EjsS* generates the Javascript code, collects auxiliary and library files,

and generates and opens an HTML page with the complete program. All at a single mouse click.

Running a simulation initializes its variables and executes the fixed relations to insure that the model is in a consistent state. The model's time evolution starts when the play/pause button in the user interface is pressed. (The play/pause button displays the ▶ icon when the simulation is paused and ⏸ when it is running.) In our current example, the program executes a numerical method to advance the harmonic oscillator differential equation by 0.05 time units and then executes the fixed relations code. Data are then passed to the graph and the graph is repainted. This process is repeated 20 times per second.

When running a simulation, *EjsS* changes its *Run* triangle icon to a red *Kill* square and prints informational messages saying that the simulation has been successfully generated and that it is running. Notice that the *EjsS* Emulator mock-up window disappears and is replaced by a new but similar Emulator window without the red border in their title. (Alternatively, you can set *EjsS*' options to run the simulation in your system's default HTML browser.) These views respond to user actions. Click and drag the particle to a desired initial horizontal position and then click on the play/pause button. The particle oscillates about is equilibrium point and the plot displays the displacement and velocity data as shown in Figure 3.9. To exit the program, close the simulation's main window.

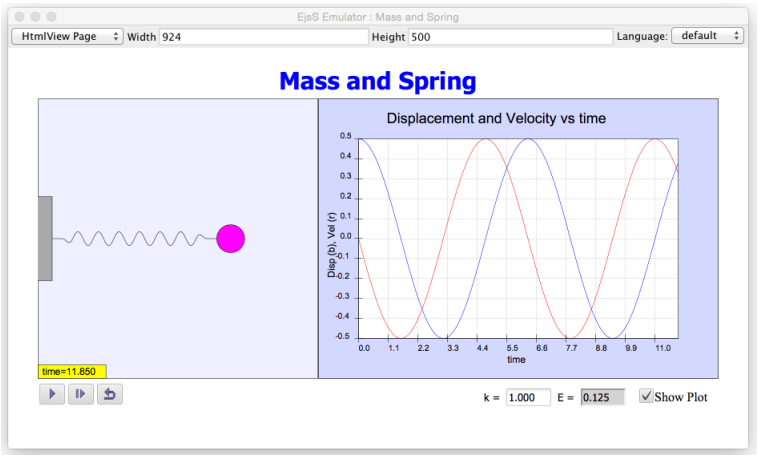

Figure 3.9: The mass and spring simulation displays an interactive drawing of the model and a graph with displacement and velocity data.

3.3 DISTRIBUTING THE SIMULATION

Simulations created with *EjsS* are stand-alone Javascript programs that can be distributed without *EjsS* for other people to use. The easiest way to do this is to package the simulation in a single executable zip file by clicking on the *Package* icon,  . A file browser appears that lets you choose a name for the self-contained jar package. The default target directory to hold this package file is the **export** directory of your workspace, but you can choose any directory and package name. The stand-alone zip file is ready to be distributed on a CD or via the Internet. Other distribution mechanisms are available by right-clicking on the icon.

Exercise 3.2. Distribution of a model
Click on the *Package* icon on the taskbar to create a stand alone zip archive of the mass and spring simulation. Copy this zip file into a working directory separate from your *EjsS* installation. Close *EjsS* and verify that the simulation runs as a stand-alone application by unzipping the file and double-clicking the **MassAndSpring.xhtml** file that is extracted from the zip file. □

An important pedagogic feature is that is very easy to distribute your simulation source code so that other people can use it with *EjsS* at any time to examine, modify, and adapt the model. (*EjsS* must, of course, be installed.) *EjsS* writes all the information in its workpanels into a small *Extensible Markup Language* (XML) description file. And it can also create a single, compressed ZIP file with that XML information and all the resource files (such as images) that you used in your simulation. This ZIP file can then be distributed, and trying to open it with *EjsS* will extract the required files from the ZIP, copy the files into a folder of your workspace, and load *EjsS* with this simulation. If a model with the same name already exits, it can be replaced. The user can then inspect, run, and modify the model just as we are doing in this chapter. A student can, for example, obtain an example or a template from an instructor and can later repackage the modified model sources into a new ZIP file for submission as a completed exercise.

Exercise 3.3. Packaging and extracting a model source files
Right click the *Package* icon and select *ZIP the simulation source files* from the menu that appears. Reopen the created ZIP file with *EjsS* and select a destination folder (different from **JavascriptExamples**) in your workspace for *EjsS* to unzip the sources. Notice that *EjsS* copies all the files in the original example for you and opens the new **MassAndSpring.ejss** XML file. □

EjsS is designed to be both a modeling and an authoring tool, and we suggest that you now experiment with it to learn how you can create and distribute your own models. As a start, we recommend that you run the mass and spring simulation and go through the activities in the second page of the *Description* workpanel. We modify this simulation in the next section.

3.4 MODIFYING THE SIMULATION

As we have seen, a prominent and distinctive feature of *Easy Java/Javascript Simulations* is that it allows us to create and study a simulation at a high level of abstraction. We inspected an existing mass and spring model and its user interface in the previous section. We now illustrate additional capabilities of *Easy Java/Javascript Simulations* by adding friction and a driving force and by adding a visualization of the system's phase space.

3.4.1 Extending the model

We can add damping in our model by introducing a viscous (Stoke's law) force that is proportional to the negative of the velocity $F_f = -b\,v_x$ where b is the damping coefficient. We also add an external time-dependent driving force which takes the form of a sinusoidal function $F_e(t) = A\,\sin(\omega\,t)$. The introduction of these two forces changes the second-order differential equation (3.1.1) to

$$\frac{d^2\,x}{dt^2} = -\frac{k}{m}\,(x - L) - \frac{b}{m}\,\frac{d\,x}{dt} + \frac{1}{m}\,F_e(t), \tag{3.4.1}$$

or, as in equations (3.1.2) and (3.1.3):

$$\frac{d\,x}{dt} = v_x, \tag{3.4.2}$$

$$\frac{d\,v_x}{dt} = -\frac{k}{m}\,(x - L) - \frac{b}{m}\,v_x + \frac{1}{m}\,F_e(t). \tag{3.4.3}$$

3.4.1.1 Adding variables

The introduction of new force terms requires that we add variables for the coefficient of dynamic friction and for the amplitude and frequency of the sinusoidal driving force. Return to the *Model* workpanel of *EjsS* and select its *Variables* panel. Right-click on the tab of the existing page of variables to see its popup menu, as in Figure 3.10. Select the *Add a new page* entry as

shown in Figure 3.10. Enter `Damping and Driving Vars` for the new table name in the dialog and an empty table will appear.

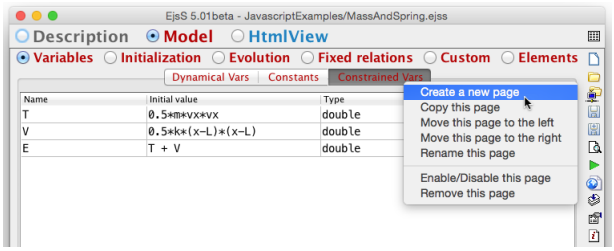

Figure 3.10: The popup menu for a page of variables.

We now use the new table to declare the needed variables. We could have used the already existing tables, but declaring multiple pages helps us organize the variables by category. Double-click on a table cell to make it editable and navigate through the table using the arrows or tab keys. Type `b` in the *Name* cell of the first row, and enter the value `0.1` in the *Initial value* cell to its right. We don't need to do anything else because the `double` type selected is already correct. *EjsS* checks the syntax of the value entered and evaluates it. If we enter a wrong value, the background of the value cell will display a pink background. Notice that when you fill in a variable name, a new row appears automatically. Proceed similarly to declare a new variable for the driving force's `amp` with value `0.2` and for its `freq` with value `2.0`. Document the meaning of these variables by typing a short comment for each at the bottom of the table. Our final table of variables is shown in Figure 3.11. You can ignore the empty row at the end of the table or remove it by right-clicking on that row and selecting *Delete* from the popup menu that appears.

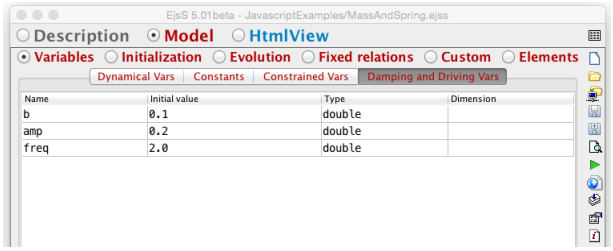

Figure 3.11: The new table of variables for the damping and forcing terms.

3.4.1.2 Modifying the evolution

We now modify the differential equations on the evolution page by adding expressions for the new terms in equation (3.4.3). Go to the evolution panel, double-click on the *Rate* cell of the second equation, and edit it to read:

```
-k/m * (x-L) - b*vx/m + force(t)/m
```

Notice that we are using a method (function) named `force` that has not yet been defined. We could have written an explicit expression for the sinusoidal function. However, defining a `force` method promotes cleaner and more readable code and allows us to introduce custom methods.

3.4.1.3 Adding custom code

The `force` method is defined using the *Custom* panel of the *Model*. Go to this panel and click on the empty central area to create a new page of custom code. Name this page *force*. You will notice that the page is created with a code template that defines the method. Edit this code to read:

```
function force (time) {
  return amp*Math.sin(freq*time); // sinusoidal driving force
}
```

Type this code exactly as shown including capitalization. Compilers complain if there is any syntax error.

Notice that we pass the time at which we want to compute the driving force to the `force` method as an input parameter. Passing the time value is very important. It would be incorrect to ask the method to use the value of the variable `t`, as in:

```
function force () { // incorrect implementation of the force method
  return amp*Math.sin(freq*t);
}
```

The reason that time must be passed to the method is that time changes throughout the evolution step. In order for the ODE solver to correctly compute the time-dependent force throughout the evolution step, the time must be passed into the method that computes the rate.

> Variables that change (evolve) must be passed to methods that are used to compute the rate because numerical solvers evaluate the *Rate* column in the ODE workpanel at intermediate values between t and $t + dt$. In other words, the independent variable and any other dynamic variable which is differentiated in the *State* column of the ODE editor must be passed to any method that is called in the *Rate* column. Variables which remain

constant during an evolution step may be used without being passed as input parameters because the value of the variable at the beginning of the evolution step can be used.

3.4.2 Improving the view

We now add a visualization of the phase space (displacement versus velocity) of the system's evolution to the *HtmlView*. We also add new input fields to display and modify the value of the damping, amplitude, and frequency parameters.

Go to the *HtmlView* workpanel and notice that the *Interface* palette contains many subpanels. Click on the tab with the icon to display the *Windows, containers, and drawing panels* palette of view elements. Click on the icon for a plotting panel, , in this palette. You can rest (hover) the mouse cursor over an icon to display a hint that describes the element if you have difficulty recognizing the icon. Selecting an element sets a colored border around its icon on the palette and changes the cursor to a magic wand, . These changes indicate that *EjsS* is ready to create an element of the selected type. (Return to the design mode –get rid of the magic wand– by clicking on any blank area within the *Tree of elements* or hitting the *Esc* key.)

Click on the `Simulation view` tree node in the *Tree of elements* as shown in Figure 3.12 to add the plotting panel to the view.

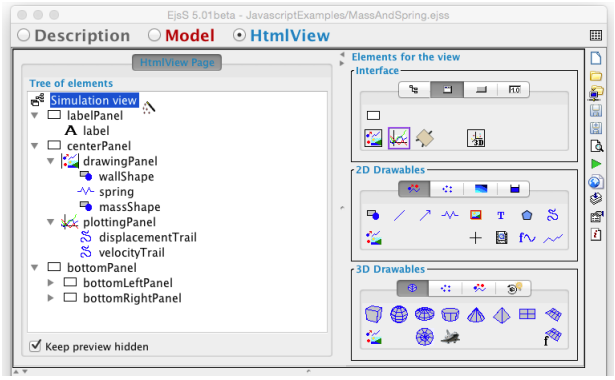

Figure 3.12: Creation of a plotting panel as a new (and last) child of the simulation view.

EjsS asks for the name of the new element and then creates the element as a new child of the simulation view (in the last position). A new plot appears but the Emulator window is too small. Resize the Emulator window

by editing the *Width* and *Height* fields at its top. This window size is useful to orient you about how the final simulation will look in devices with fixed screen resolution (such as tablets). Finally, edit the properties table of the newly created plotting panel element to set the `Title` property to `Phase Space`, the `TitleX` property to `Displacement`, and the `TitleY` property to `Velocity`. (*EjsS* will add leading and trailing quotes to these strings to conform to the correct Javascript syntax.) Set the minima and maxima for both X and Y scales to `-1` and 1, respectively, and leave the other properties untouched.

The plotting panel is, as its name suggests, the container for the phase-space plot. Phase space data are drawn in this panel using a 2D element of type `Trail`, . Find the `Trail2D` element in the `2D Drawables` palette and follow the same procedure as before. Select the `Trail` element and create an element of this type by clicking with the magic wand on the new phase space plotting panel. Finally, edit the properties of the new trail element to set its `InputX` property to `x - L` and its `InputY` property to `vx`. This assignment causes the simulation to add a new `(x - L,vx)` point to the trace after each evolution step, thus drawing the phase-space plot shown in Figure 3.13.

Figure 3.13: The new phase-space plot added to the simulation view.

To finish the modifications, we will add a new panel before the new plotting frame that shows the sinusoidal driving force parameters.

- Select the `Panel` element icon, ▭, on the *Windows, containers, and drawing panels* subgroup of the *Interface* palette. Click with the magic wand on the `Simulation view` root node within the *Tree of elements* to create a new panel named `forceParamPanel` as its last child. Right-click and use the *Move up* option to locate it before the phase space plotting panel. (You can also drag-and-drop it to the new position in the tree.)
- Select the `Label` element icon, **A**, on the *Buttons and decorations* subgroup of the *Interface* palette and create a new element of that type in the `forceParamPanel` panel. Set the label's text property to

`"frequency ="`.

- Select the `Field` element icon, , and create a new element named `freqField` in the force parameter panel. Edit the `freqField` properties table as shown in Figure 3.14. The connection to the `freq` variable is established using the `Value` property. Click on the second icon to the right of the property field, , and choose the appropriate variable. The variable list shows all the model variables that can be used to set the property field. The `Format` property indicates the number of decimal digits with which to display the value of the variable.
- Repeat this process to add the `amp` variable to the user interface.

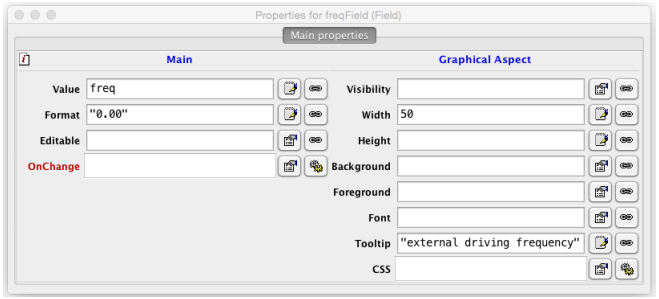

Figure 3.14: The table of properties of the `freqField` element.

3.4.3 Changing the description

Now that we have changed the model and the view, we should modify the description pages of our simulation. Go to the *Description* workpanel and click on the tab of the first page, the one labeled `Introduction`. Once you see this page, click the *Click to modify the page* icon, . The description page will change to edit mode, as shown in Figure 3.15, and a simple editor will appear that provides direct access to common HTML features.

If you prefer to use your own editor, you can copy and paste HTML fragments from your editor into the *EjsS* editor. If you know HTML syntax, you can enter tagged (markup) text directly by clicking the source icon, , in the tool bar. You can even import entire HTML pages into *EjsS* by clicking the *Link/Unlink page to external file* icons, , .

Edit the description pages as you find convenient. At least change the discussion of the model to include the damping and driving forces. When you are done, save the new simulation with a different name by clicking the *Save as* icon of *EjsS*' taskbar, . When prompted, enter a new name for your simulation's XML file. The modified simulation is stored in the **MassAndSpringComplete.ejss** file in the **source** directory for this chapter.

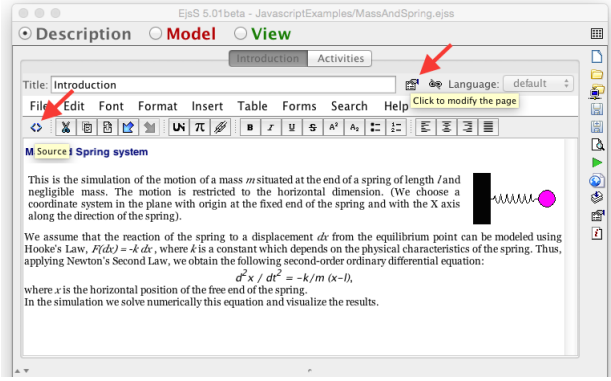

Figure 3.15: The HTML editor of *EjsS*. The added red arrows point to the edit and source code edition mode icons.

3.5 PROBLEMS AND PROJECTS

Problem 3.1 (Energy). Add a third plotting panel to the dialog window of the **MassAndSpringComplete.ejss** simulation that will display the evolution of the kinetic, potential, and total energies.

Problem 3.2 (Function plotter). The analytic solution for the undriven simple harmonic oscillator is

$$x(t) = A\sin(w_0 t + \phi) \tag{3.5.1}$$

where A is the amplitude (maximum displacement), $w_0 = \sqrt{k/m}$ is the natural frequency of oscillation, and ϕ is the phase angle. Consult a mechanics textbook to determine the relationship between the amplitude and phase angle and the initial displacement and velocity. Use the **FunctionPlotter.ejss** simulation in the examples directory to compare the analytic solution to the numerical solution generated by the **MassAndSpringComplete.ejss** model.

Project 3.1 (Two-dimensional oscillator). Modify the model of the mass and spring simulation to consider motion that is not restricted to the horizontal direction. Assume that a second spring with spring constant k' produces a vertical restoring force $F_y(\delta y) = -k'\,\delta y$. Modify the simulation to allow the user to specify the Hooke's law constants as well as the initial conditions in both directions. Describe the motion produced without a driving force but under different initial conditions and with different spring constants. (Try $k = 1$ and $k' = 9$.) Show that it is possible to obtain circular motion if $k = k'$.

Project 3.2 (Simple pendulum). Create a similar simulation as the one described in this chapter for a simple pendulum whose second-order differential

equation of motion is

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin(\theta), \tag{3.5.2}$$

where θ is the angle of the pendulum with the vertical, g is the acceleration due to gravity, and L is the arms's length. Use fixed relations to compute the x and y position of the pendulum bob using the equations:

$$x = L\ \sin(\theta)$$
$$y = -L\ \cos(\theta).$$

Chapter Four

Converting from Java to Javascript

> Everything must change so that everything can remain the same
> *Giuseppe di Lampedusa in 'The Leopard'*

In this chapter, we go through the process of porting an existing Java simulation created with *EjsS* into an equivalent Javascript simulation. The *EjsS* workpanel architecture makes this conversion easy, the *HtmlView* being the only part that needs to be re-created from scratch. We also show how the *HtmlView* can be enhanced using cascading stylesheet (CSS) properties.

4.1 PORTING A SIMULATION

We choose an old friend, the **MassAndSpring.ejs** Java-based simulation studied in Chapter 2 and plan to convert it into a pure Javascript one. The complete process consists of the following steps:

1. Loading the **.ejs** file and saving it as a **.ejss** file.
2. The *Description* remains basically untouched.
3. In the *Model* tab, we need to edit the Java code in different editors and convert it to equivalent Javascript code.
4. The Java Swing-based *View* disappears and must be replaced by an equivalent (or similar) HTML-based *HtmlView*

We describe in the next sections each of these steps to convert the **MassAndSpring.ejs** Java simulation into Javascript. The reciprocal process (that is, converting from Javascript to Java) is also possible and consists of the same steps, if only in the opposite direction.

4.2 CREATING THE NEW FILE FROM THE OLD ONE

We begin by creating a new **MassAndSpring.ejss** file from the original **MassAndSpring.ejs** one. Run the Javascript flavor of *EjsS*. Important,

the **Javascript** flavor!

> If you happen to have already a Java flavor of *EjsS* open, you can close it, you will not need it — for the time being.

Now, ask *EjsS* to load the original **MassAndSpring.ejs** file. *EjsS* will recognize the clash between the programming language supported by the current flavor, and the **.ejs** extension of the file (which indicates a Java *EjsS* file). The dialog shown in Figure 4.1 will warn you of this and will ask you what to do about it.

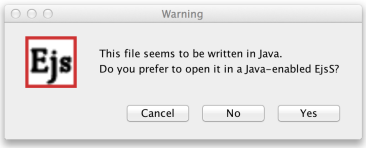

Figure 4.1: Dialog caused by a possible programming language conflict.

The options provided by this dialog are the following:

Cancel : This will stop loading this file altogether.

No : This will ignore the conflict and will load this file.

Yes : This will launch a new instance of *EjsS* with the correct programming language support and will load this file in it.

You need to select **No** for the purposes of this chapter. This will open the original Java file in this Javascript instance of *EjsS*. You will notice that the *Description* and *Model* tabs of *EjsS* are loaded with the information of the **MassAndSpring.ejs** file, but the *HtmlView* tab will remain empty.

Before we do any changes, and in order to preserve the original file, we want to save the file with the extension for Javascript files, **.ejss**. Click now the *Save as* icon of *EjsS*' taskbar,  , (preferably) change to a different directory of your workspace **source** directory, and change the name of the output file to **MassAndSpring.ejss**. That is, just change the file extension to **.ejss**. This extension identifies the new file as a Javascript *EjsS* file.

Now, *EjsS* will proceed to save the new file and will detect that this file has a Java view and ask you whether you want to save it or not. See Figure 4.2.

Your answer should be **No**. We won't need the Java view in our new Javascript simulation. (Recall that, since we chose a different extension

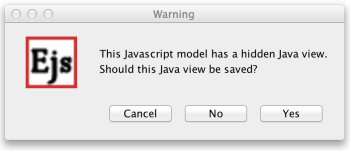

Figure 4.2: Dialog concerned about the existing Java view.

for the new file, the original **MassAndSpring.ejs** file remains unchanged.) *EjsS* will have saved the new file and, if necessary, copied any auxiliary files required by this simulation (if you changed the target directory), thus creating a completely independent new simulation file. We are now ready to start the transition to a pure Javascript simulation.

4.3 CHANGES TO THE DESCRIPTION

The good news for the *Description* is that no changes are really needed. The Description are just HTML pages that will work just fine for the Javascript model.

If only, we recommended that you review your HTML files to make sure the HTML code in them is correctly formed. The reason is that, if you later want to create an ePUB document with your Javascript simulation, *EjsS* needs the HTML code to be correct. More precisely, ePUB documents require XHMTL files (which are basically more strictly enforced HTML files) and *EjsS* can either take XHTML files or try to convert your (correctly formed) HTML files for it.

Typical editions required to make an HTML file correct include the following:

- Make sure all HTML opening tags have their corresponding closing tag. In particular, paragraphs tags $< p >$ need a matching $< /p >$.
- Make sure all HTML single tags have a closing /, as in $< br/ >$.

However, the best recommendation we can make to you, if you plan to create ePUB documents with your description pages in them, is that you create stand-alone valid XHTML file (you can validate them with free tools available on the Internet) and use the *Description* feature that lets you link your pages to external HTML or XHTML pages (see Figure 4.3).

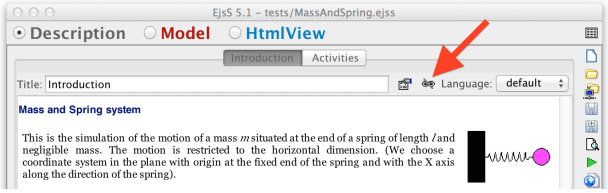

Figure 4.3: Icon to link a description page to an external HTML or XHTML file.

4.4 CHANGES TO THE MODEL

The model needs that you change all Java code to Javascript code. Let's examine in detail each of the parts of the model for our sample simulation and do the required changes.

4.4.1 Variables

Javascript has no types for variables. All variables are declared with a *var* keyword. This means that, in principle, Javascript makes no difference among integers, doubles, Strings, etc.... But it does! For instance, you should not use a double variable as index for an array.

For this reason, and also because it helps clarify the use of variables in your model (what values they can have and where you can or cannot use them), we have left the feature that forces you to assign a type to a variable. The types accepted are the basic ones, and no more.

If, for any reason, your Java model has variables declared to be of any non standard type (java.awt.Color is a typical example of this), you must convert it to one of the predefined types. (By the way, colors are indicated in HTML5 using strings.) If your model uses sophisticated Java objects you may have created or imported from an external library, then you are in trouble. Again, you must resort to using only basic types.

The initial values you can assign to basic variables are very much like what you can assign them in Java. (Arrays an objects are an exception, see below.) For this reason, simple models do not require *any* change at all in the tables of variables, when changing them from Java to Javascript. In particular, our example needs no change to the *Variables* part of the model.

Arrays

Arrays are declared somewhat differently in Javascript. The editor for variables in *EjsS* takes care of this and you need only to indicate the dimensions, as you would do in the Java flavor. However, giving an initial value to an array is different, since Javascript uses square brackets as delimiters in an array, while Java uses braces. Hence, an initial value like this in Java:

```
double[][] myArray = new double { {1.0,2.0,3.0}, {3.0,4.0,5.0} };
```

should be written as follows in Javascript:

```
var myArray = [ [1.0,2.0,3.0], [3.0,4.0,5.0]];
```

Also, resizing an array is different in Javascript. The Java syntax:

```
doubleArray = new double[n][m];
```

converts in Javascript to:

```
doubleArray = new Array(n);
for (var i=0; i<n; i++) doubleArray[i] = new Array(m);
```

(assuming `n` and `m` are valid integers). A final difference is that Javascript does not guarantees to initialize to 0 the elements of the allocated array, nor check for incorrect use of indexes.

Objects

Javascript supports the concept of an *Object*. Basic objects are just dictionaries of other variables. You add variables to objects as follows:

```
var myObject  = { };
myObject.name = "My object";
myObject.value = 3.0;
```

The use you make of these variables is up to you. Sophisticated use of Javascript objects is allowed, though out of the scope of this manual.

Variables not declared

Finally, Javascript is a loose language and does not force you to declare variables before using them. For instance, the code

```
n = 3;
_println ("n = "+n);
```

will produce the intended result, even if `n` has not been declared elsewhere. This may lead to undesired results, if you happen to type incorrectly one of your variables. (Javascript will think it is a new one!) Watch for this caveat. *EjsS* tries to help you with this problem and incorporates a *Lint* processor [1] that looks for potential errors in your code, including this one. In occasions like the one described here, the *EjsS Output area* will display a message like the following, when you try to run the simulation:

```
Lint error: 'n' is not defined.
     n=3;   // > Initialization.Init Page:1
```

4.4.2 Pages of code

All pages of code in the *Initialization*, *Evolution*, *Fixed relations* and *Custom* must be changed from Java to Javascript syntax. Fortunately, the syntax of algorithms for both programming languages is very similar. The main changes usually required for code in these parts of the model are the following:

- Use `var` to declare local variables, instead of `int`, `double`, etc. A typical place for this is in `for` loops.
- The scope of local variables is that of the block in which they are used. Even if you declare them *after* they are used. For this reason, it is recommended to declare all variables for once at the beginning of each page of code.
- Function declaration in Javascript includes no information about the return type or the type of the parameters. A declaration of a Java function like this:

  ```
  public double force (double time) {
  ```

[1] <*http://en.wikipedia.org/wiki/Lint_(software)*>

```
  return amp*Math.sin(freq*time);
}
```

turns into the following Javascript code:

```
function force (time) {
  return amp*Math.sin(freq*time);
}
```

Our sample simulation requires no change of its *Fixed relations* page of code, since the Java expressions:

```
T = 0.5*m*vx*vx;
V = 0.5*k*(x-L)*(x-L);
E = T + V;
```

are also valid Javascript expressions.

ODE editor

The use of the ODE editor, for evolution pages that require solving ordinary differential equations, is straightforward. The only obvious change is that expressions in the rate equations and code in preliminary code, events, and other pages of code, need to use valid Javascript syntax. The Javascript ODE editor implements, for the time being, less solvers than its Java counterpart. But the underlying code and mechanisms are exactly the same.

4.4.3 Model elements

The Javascript flavor of *EjsS* includes also the concept of *Model elements*, as a way to access third party Javascript libraries. However, the elements provided by the Java and Javascript flavors are completely different and there is no attempt to make this implementation converge.

Model elements used by your simulation, should your original Java simulation use them, are preserved to make you conscious of their need for your model. But you must immediately remove them and replace them by equivalent Javascript model elements, or by your own coding.

In other words, porting from Java to Javascript (or viceversa) simulations that depend on the use of Model elements can be extremely difficult.

4.5 IMPLEMENTING AN HTML VIEW

One part of the simulation that needs to be completely created from scratch is the *HtmlView*. The reason is that creating graphical user interfaces with Swing (the Java library for interfaces) and HTML5 is based on completely different principles. To name one, Swing is a window-oriented platform, while HTML5 is designed to be displayed inside a web browser in a single window.

For this reason, we decided to create a number of pure HTML5 elements, based on our past experience with Java, but adopting a newer approach, more natural to HTML5 and not restricted by backwards compatibility issues. The result is a collection of HtmlView elements that you combine (in a similar way to how you combined View elements) to create interfaces of great quality and with the same flexibility as with the Java version.

An important difference with the Java flavor is that a simulation can have more than one HtmlViews. Although many authors typically only create one, we have provided this feature for situations where you want to plan to support devices with different sizes and orientations. Each HtmlView you create has a preferred with and height that you specify. In runtime, if you provided more than one HtmlViews, the simulation will use the one that best matches the device' screen size and orientation.

In our case, we will create only one HtmlView that resembles the one of the original Java version. So, click on the empty HtmlView panel to create an initially empty HtmlView.

Now, it is a very good idea to launch a Java flavored instance of *EjsS* and load into it the original **MassAndSpring.ejs** file. The reason is that we want to have both instances of *EjsS*, the Java and the Javascript flavors, side by side, and inspect the Tree of View elements in the former, as we create and customise the Tree of HtmlView elements in the latter. Figure 4.4 shows both views (the Javascript HtmlView still empty), side by side.

Your task consists now of creating the HtmlView by selecting the elements provided by the HtmlView palette, in a way that mimics the selection done by the Java version. You will need to customise each HtmlView element's properties to match the use of the model variables and actions that was done in the original Java view. (Recall to write valid Javascript code for the action properties!)

A good way to speed up the process is to make use of Custom elements

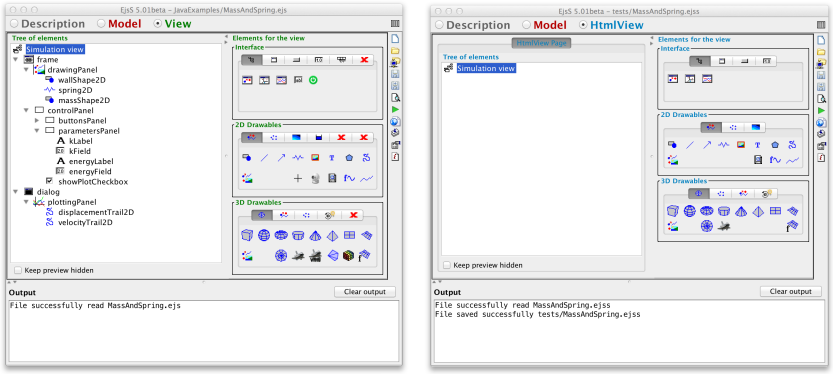

Figure 4.4: Java tree of elements (left) and empty HTML5 tree of elements.

(provided in the first tab of the *Interface* palette. These consist of a selection of predefined combinations of HtmlView elements that are common to many simulations.

The creation of a new HtmlView, element by element, is, perhaps, the most time-consuming part of the process of porting an existing Java simulation to Javascript. We don' t get into details here, but you can see the final result of our work in the **JavascriptExamples/MassAndSpring.ejss** file included in your workspace by the standard distribution of *EjsS*.

One important feature of HtmlView interface elements is the presence of a *CSS* property. This property lets you set directly the CSS characteristics of the underlying HTML5 element. CSS (Cascading style sheets) is a complex and flexible environment to help create very nice interfaces, and we do not try to cover it in detail. But even a small amount of CSS will help you improve your HtmlViews. [2]

Another feature of the implementation of HtmlView element properties is that you can very easily set any of them programatically. Just use a sentence of the form:

```
_view.elementName.setProperty("PropertyName",value);
```

where `elementName` is the name you gave to the element, `PropertyName` is the name of the property as it appears in the property edition dialog, and `value` is the value you want to give to the property. Examples of this are:

```
_view.plottingPanel.setProperty("Display","none"); // Hide it
```

[2] See for instance <*http://www.w3schools.com/css/*>.

```
_view.bottomPanel.setProperty("Width",450); // Resize it
```

4.6 PAGE LAYOUT WITH CSS

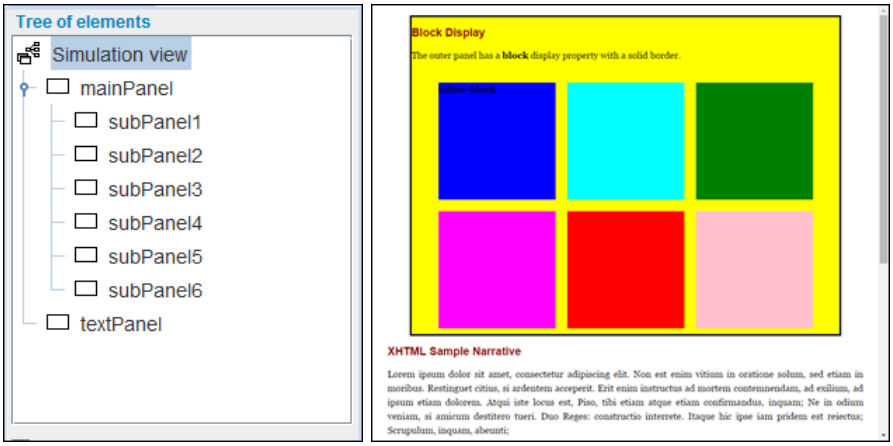

Figure 4.5: The inline-block display value places panels side-by-side and wraps at the edge of the view.

It is straightforward to create a standard flow layout using cascading stylesheet (CSS) properties as shown Figure 4.5. A Panel Element in the simulation's HTML view is an HTML `<div>` that can be modified using inspector's css field. For example, we set the display, vertical alignment, and margin properties of panels in the the Flow Layout example in the templates directory of the standard *EjsS* distribution with the following entry.

```
{"display":"inline-block","vertical-align": "top","margin":"10px"}
```

Common `display` values are `block`, `inline`, and `inline-block`. A block display creates a viewing region with an upper left hand corner location below the previous block so that a sequence of block displays will align vertically in the view. A `display:inline` selector places elements side-by-side from left to right similar to words on page. An `display:inline-block` selector places panels side-by-side and wraps at the edge of the view while respecting the panel's width and height properties.

Examine the Flow Layout example [3] and note how the CSS field is used to format the main panel to create a display with a solid border and custom margins.

[3] All examples in this section can be found in the **CSSTemplates** subfolder of the **JavascriptExamples** sample directory

```
{"display":"block",
 "border":"solid",
 "margin-left":"1.0cm",
 "margin-right":"1.0cm",
 "margin-top":"0.0cm",
 "margin-bottom":"0.25"}
```

Run this model in a web browser (not the *EjsS* emulator, so that you can resize the browser window) and observe how the panel resizes and how the subpanel locations change as the browser window size is adjusted. Change the CSS properties for the main panel, the text panel, and the six subpanels and observe the effect in the view.

Exercise 4.1. CSS Properties
Modify the CSS field value of the first subpanel in the Flow Layout example.

```
{"display":"inline-block",
 "vertical-align": "top",
 "margin":"10px",
 "width":"100px",
 "height":"300px"}
```

Setting width and height properties using the inspector's css field has the same effect as setting the width and height using inspector fields. Which of these width and height values does the simulation respect if both are present? □

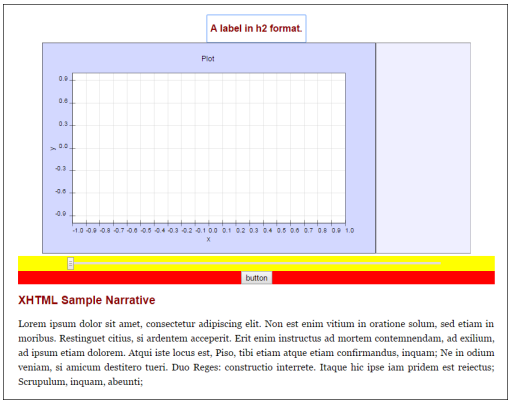

Figure 4.6: Two plots are displayed inline with different width properties.

A panel's width and height property values can include units such as px (pixel), in (inch), or cm (centimeter) or it can be a % of the HTML view. Examine the CSS properties in the Two Plot Horizontal Layout example and note how the panel widths are initialized in the Display Vars table. It

is sometimes convenient to set size properties based in the platform because mobile and e-book displays are smaller than desktop computer screens. In this example, we set different panel sizes in the Display Vars table after testing the `_isMobile` and `_isEPub` flags. When running in a desktop browser the panel widths are set to 70% and 20% as shown in Figure 4.6 but they are set to 300 pixels and 100 pixels when running on a mobile platform. Run the example and note how the view changes as the browser window is resized.

The Two Plot Vertical Layout example (not shown here) implements an often used plotting, drawing, and control panel layout. The plotting and drawing panels are placed in a container panel that has width and height properties but note that the interior panels have a width of 100% so that they completely fill the container which has a width of 70%.

The CSS position selectors allows us to position an element anywhere in the view and Relative Position example demonstrates how this is done. Elements can be positioned using the top, bottom, left, and right properties. However, these properties will not work unless the position property is set first and they work differently depending on the positioning method. Examine the Tree of Elements in the Relative Position example that there are three top-level panels arranged in a block layout. The `simPanel` has a CSS property that renders the `containerPanel` and `controlPanel` elements as an inline block on the same page.

```
{ /* Contents should display on same page */
  "page-break-inside":"avoid",
  "display":"inline-block"
}
```

Relative positioning is performed using the CSS position selector in the `containerPanel`. The selector selector value is *static by default and we must first set the position value to* relative, *fixed, or* absolute to position elements within the container. A relative positron renders the container in the normal flow and allows us to establish a coordinate origin for placing panels within the container.

```
{ /* Establish origin for absolute positioning*/
  "display":"block",
  "position":"relative",
  "top":"0px",
  "left":"0px"
}
```

The plotting panel is rendered within the container panel in its normal

position with a width set to 100% so that it completely fills the container as was done in the Two Plot Vertical Layout example. But the drawing panel uses an absolute position value to position render the panel in the center of the container using CSS top and left selectors. The check box in the control panel invokes the `switchView ()` function in its `onChange` action to set these selectors.

```
function switchView () {
  if (showDrawing) {
    _view.drawingPanel.setProperty("CSS", {
      "display" : "block",
      "position":"absolute",
      "top":"25%",
      "left":"25%"
    } );
  } else {
    _view.drawingPanel.setProperty("CSS", {
      "display" : "none"
    } );
  }
}
```

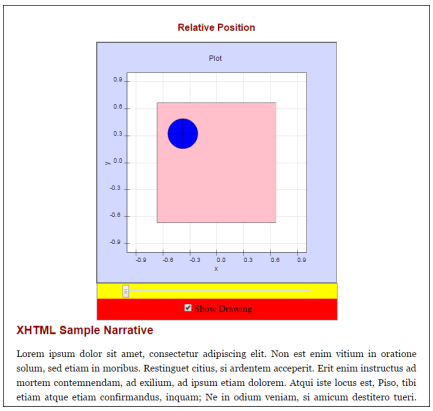

Figure 4.7: A plotting panel with a centered drawing panel that can be hidden.

The CSS selectors allow us to position an element element, to overlap elements, and to specify what should happen when an element's content is too big as show in Figure 4.7. An element with a relative position property is rendered in the normal flow of the page unless we specify the top and left selectors. An element with absolute position is rendered relative to the first parent element that has a position other than static. An element with fixed position is rendered relative to the browser window. It will not move even if the browser window is scrolled.

4.7 CONCLUSION

As conclusion for this chapter, we just repeat what we said at its beginning. Porting an existing Java simulation to a pure JavaScript one is rather straightforward in *EjsS*, and consists of the following steps:

1. Loading the **.ejs** file and saving it as a **.ejss** file.
2. Review the *Description* pages for HTML correctness.
3. Edit the Java code in different editors of the *Model* to convert it to equivalent Javascript code.
4. Create a HTML-based *HtmlView* that mimics the original Java based *View*.

Creating the HtmlView is, by far, the most time-consuming part of the process.

Appendix A

Adding "Events" in Easy Java/Javascript Simulations

Adapted and edited by Francisco Esquembre

This document provides a description of "events" in EJS—both what they are and how they are added.

A.1 WHAT IS THE PURPOSE OF AN "EVENT" IN EJS?

The purpose of an "event" is to provide a way for a certain *action* to occur (for example, pausing the simulation) at the exact instant that a certain *condition* is met (for example, a ball hitting the ground). For the example of a falling ball, the following lines of code can be used to implement the Euler algorithm:

```
[8]
y = y + v*dt; // Update the height, y (using the velocity)
v = v + a*dt; // Update the velocity, v (using the acceleration)
t = t + dt;   // Update the time, t (using the time step, dt)
```

For the example of a ball *hitting the ground*, the "event" is analogous to the following lines of code:

```
[8]
if (y < 0)  // If the condition is true...
{           // then the code within {...} is executed
  _pause(); // This pauses the simulation.
}
```

Note these lines of code pause this simulation when the ball hits the ground. (More precisely, this pauses the simulation *after* the ball hits the ground.) Using an event will accomplish the same purpose, but the use of an event will be superior to these lines of code in two ways:

i) The event will be more accurate than this code.

ii) The event will be even simpler to create than this code.

Using an event will be more *accurate* than the lines of code listed above for the following reason. The lines of code above are only executed at the discrete time steps $t = 0$, Δt, $2\Delta t$, $3\Delta t$, $4\Delta t$, etc. When using an event, EJS does much better than this. If the height is found to change from a positive value to a negative value between $t = 2\Delta t$ and $t = 3\Delta t$, the ODEs will be re-evaluated (meaning that the height will be re-evaluated) at $t = 2.5\Delta t$ (midway between the two times). This will hence provide a better approximation to the *actual* time that the ball hits the ground. This process will then be repeated. If the height is found to change from positive to negative between $t = 2\Delta t$ and $t = 2.5\Delta t$, the ODEs will be re-evaluated at $t = 2.25\Delta t$ (the new midway point between the times). The number of times that this process is repeated is referred to as the number of *iterations*, and since each iteration involves cutting the time interval in half, this method is referred to as the *bisection* method. After several iterations, this process should give a very accurate approximation to the exact moment that the event occurs. (This process of finding precisely where a function crosses zero is generally referred to as *root finding*.)

A.2 HOW ARE EVENTS ADDED IN EJS?

Creating an event in EJS is also very *simple* to do. After creating a page of ODEs, an event can be added by clicking on the button labeled "Events". After creating an event, a window will appear. This window is shown in Fig. A.1.

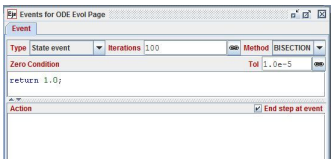

Figure A.1: The blank page that appears when adding an event.

At the top of the window shown in Fig. **??** several options can be specified. The menu in the top left corner allows you to choose the "type" of event to be used, as described in the following section. The maximum

number of iterations (described in the previous section) can also be specified at the top of the page. In the top right corner, the method of iteration can be selected. The bisection method is described in the previous section. The secant method is similar, but uses a secant line between the two points in time to estimate when the crossing event occurs.

Finally, it is necessary to specify the "Zero condition" and the "Action". The zero condition consists of one or more lines of code, and it must include a `return` statement. The quantity (or the **variable**) following the word "return" will be checked to see whether or not the event has occurred (e.g., whether or not the ball has hit the ground). For example, with the statement `return 1.0;` the event will never occur because 1.0 will always be greater than zero! In fact, you would *never* want to return a fixed numerical value because a fixed value can never change sign. Hence, you need to replace the "1.0" with the thing that *does* cross zero (and does change sign) at the moment that the event should occur. One of the keys to using an event is to decide what quantity should follow the word `return`. The "Action" is the code to be executed when the event occurs, and it needs to be entered into the lower part of the window.

For the example of a ball hitting the ground, the *event* can be described as "the moment that the height crosses the value $y = 0$". Hence, the `return` statement would be `return y;`. This completed event is shown in Fig. A.2.

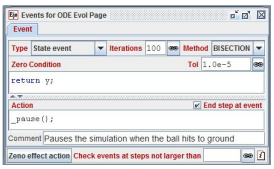

Figure A.2: The completed event that will pause the simulation when an object hits the ground.

A.3 DIFFERENT "TYPES" OF EVENTS

In the upper-left corner of the Event window (visible in Figs. **??** and **??**), you are able to choose between three different "types" of events:

i) State event

ii) Zero crossing

iii) Positive crossing

Specifying the "type" of event to use can be a bit confusing for the following reason: Your choice of which type of event to use *doesn't matter*... unless it does! For example, the event shown in Fig. **??** that pauses the falling ball will work correctly regardless of which type of event is used. In other situations though, the correct choice of event type can be very important.

Let us first consider the "Zero crossing" event. This does exactly what you would expect. When the value of the quantity specified by the `return` statement crosses zero, the event is triggered, and the *Action* is executed. The "Positive crossing" event does the same thing, except this type of event is *only* triggered when the value of the quantity changes from positive to negative—not negative to positive. Clearly either of these would work for the falling object, since—at the moment that it hits the ground—the height crosses zero *and* changes sign from positive to negative. As a different example, when using an event to measure the period of some type of oscillation, you might wish to use a *positive* crossing rather than a *zero* crossing. This would allow you to measure the *complete* period rather than only half an oscillation. You might wonder why there is not a corresponding "Negative crossing". Detecting a negative crossing is a very reasonable thing to want to do, and it is very easily accomplished: Simply use a positive crossing, and include a negative sign after the word `return` in the Zero Condition!

The "State event" is just like the "Positive crossing" event, except that a *state* event should be used in situations where it is *not allowed* for the value of the quantity to be negative. Examples could include objects *bouncing* off of the ground, or off of a wall, or off of each other. In these situations, you are *required* to include code within the Action that will keep the quantity (following the `return`) out of the "forbidden region". For the examples of bouncing objects, this is accomplished by simply changing the sign of the velocities within the action. For a concrete example of the difference between state events and positive crossing events, do the following:

i) Open the Event window shown in Fig. **??**, and remove (or comment out) the line of code that appears in the Action. (Now the program will continue after the event is triggered.)

ii) Run the program with *State event* selected for the event type. With a state event, negative values are not allowed, so you will see an error. (This is EJS politely tapping you on the shoulder to let you know that you are "breaking the rules".)

iii) Run the program again with either *Zero crossing* or *Positive crossing* selected for the event type. Now there is no error, since negative values *are* allowed for these types of events.

As a final example, again select "State event," but now specify `v = -0.9*v;` for the Action. This will make the ball *bounce*; but since the coefficient of restitution is less than 1, the maximum height of the ball will decrease (by the *same fraction*) with each bounce. We could now ask the question, "How long will it take for the ball to come to rest?" which is reminiscent of *Zeno's paradoxes*. To address these situations, click on the "Zeno effect action" button shown in the lower-left corner of Fig. **??**, and enter additional code to be executed in the event of infinitesimal bounces—e.g., pausing or resetting the simulation.